

Threads

- Recap:
 - Why Threads?
 - User-Level Threads vs. Kernel-Level Threads
 - Thread-Based vs. Event-Based System Design?
 - Event-Based: John Ousterhout, "Why Threads are a Bad Idea (for most Purposes)"
 - Thread-Based: von Beren, Condit, Brewer, "Why Threads are a Bad Idea (for high-concurrency Servers)"
 - Example: Windows Completion Ports.
 - Reading: Silberschatz, Ch 3 & 4.
-

Why Threads?

- Many interactive applications run in loops.
 - For example, an interactive game.
- ```
while (1) {
 /* Read Keyboard */
 /* Recompute Player Position */
 /* Update Display */
}
```
- Reference [B.O. Gallmeister, "POSIX.4, Programming for the Real World," O'Reilly&Assoc., Inc.]
-

## Why Threads?

- Many interactive applications run in loops.
- For example, an interactive game.

```
while (1) {
 /* Synchronize to Highest
 Frequency */
 /* Read Keyboard */
 /* AND Read Mouse */
 /* Recompute Player Position */
 /* Update Display */
 /* AND emit sounds */
}
```

- Reference [B.O. Gallmeister, "POSIX.4, Programming for the Real World," O'Reilly&Assoc., Inc.]

## Why Threads?

- Many interactive applications run in loops.
- For example, an interactive game.

```
while (1) {
 /* Synchronize to Highest
 Frequency */
 /* Read Keyboard */
 /* AND Read Mouse */
 /* Recompute Player Position */
 /* Update Display */
 /* AND all other lights */
 /* AND emit sounds */
 /* AND more sounds */
 /* AND move game physically */
}
```

- **It ain't over yet!**
- **What about compute-intensive operations, like AI, video compression?**
- **How about Signal Handlers?**

Suddenly, application is getting complex!

- Reference [B.O. Gallmeister, "POSIX.4, Programming for the Real World," O'Reilly&Assoc., Inc.]

## Example: Reading the Mouse

```

while (1) {
 /* Synchronize to Highest
 Frequency */
 /* Read Keyboard */
 /* AND Read Mouse */
 /* Recompute Player Position */
 /* Update Display */
 /* AND all other lights */
 /* AND emit sounds */
 /* AND more sounds */
 /* AND move game physically */
}

```

Separate Thread

```

read_mouse() {
}

```

## Reading the Mouse: Thread Creation

```

while (1) {
 /* Synchronize to Highest
 Frequency */
 /* Read Keyboard */
 /* AND Read Mouse */
 /* Recompute Player Position */
 /* Update Display */
 /* AND all other lights */
 /* AND emit sounds */
 /* AND more sounds */
 /* AND move game physically */
}

```

```

/* The Mouse Input Function */

void * read_mouse() {
 char buf[BUFSIZE]; ssize_t nbytes;
 for (;;) {
 if ((nbytes = read_from_mouse(buf, BUFSIZE)) <= 0)
 break;
 dosomething_with(buf, nbytes);
 }
 return NULL;
}

```

## Reading the Mouse: Thread Creation (II)

```

#include <pthread.h>

int error;
pthread_t tid;

if (error = pthread_create(&tid, NULL, read_mouse, NULL))
 perror("Failed to create read_mouse thread");

while (1) {
 /* Synchronize to Highest
 Frequency */
 /* Read Keyboard */
 /* AND Read Mouse */ <- Handled by separate thread!
 /* Recompute Player Position */
 /* Update Display */
 /* AND all other lights */
 /* AND emit sounds */
 /* AND more sounds */
 /* AND move game physically */
}

```

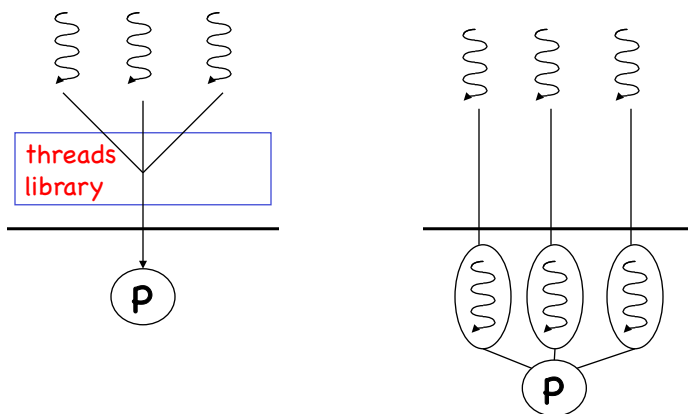
```

f, BUFSIZE)) <= 0)

```

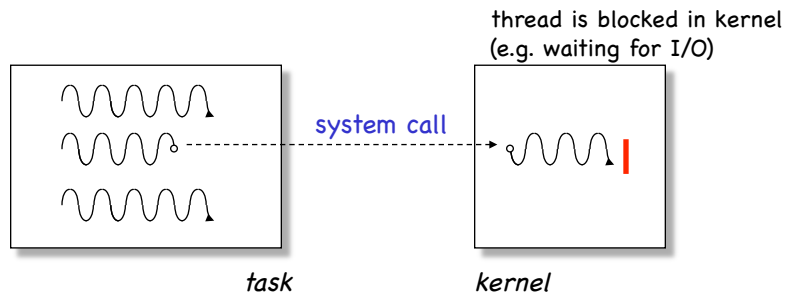
## User-Level vs. Kernel-Level Threads

- **User-level:** kernel not aware of threads
- **Kernel-level:** all thread-management done in kernel

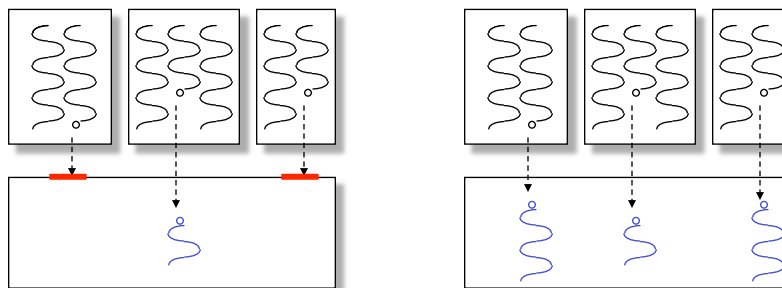


## Potential Problems with Threads

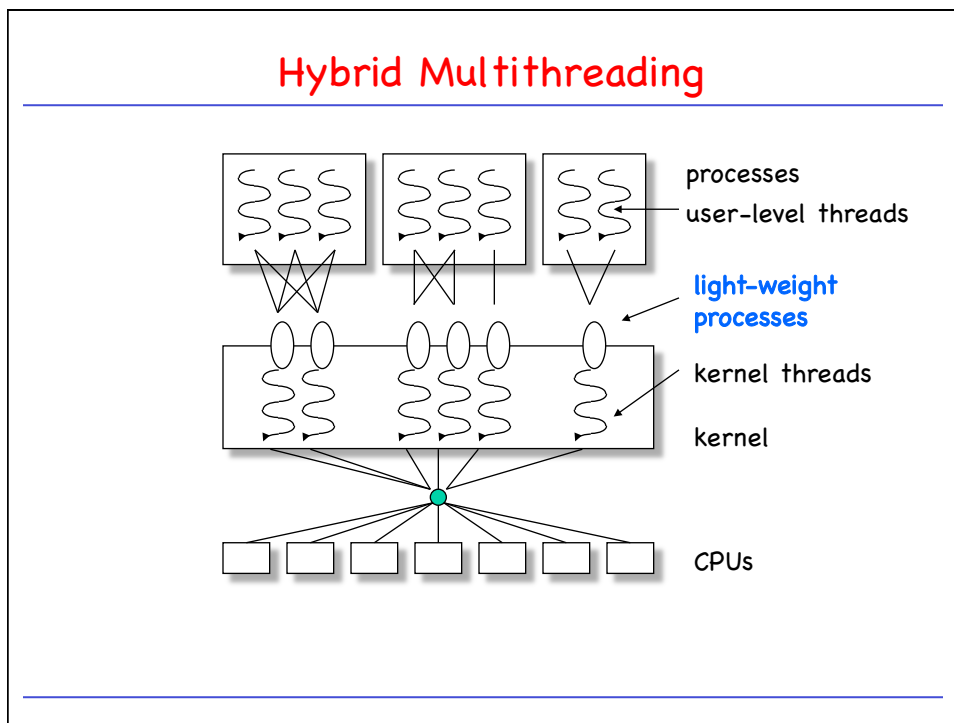
- General: Several threads run in the same address space:
  - Protection must be explicitly programmed (by appropriate thread synchronization)
  - Effects of misbehaving threads limited to task
- User-level threads: Some problems at the interface to the kernel: With a single-threaded kernel, as system call blocks the entire process.



## Singlethreaded vs. Multithreaded Kernel



- Protection of kernel data structures is trivial, since only one process is allowed to be in the kernel at any time.
- Special protection mechanism is needed for shared data structures in kernel.



### Threads in Practice: Issues in Server Software Design [Comer]

- **Concurrent** vs. **Iterative** Servers:
 

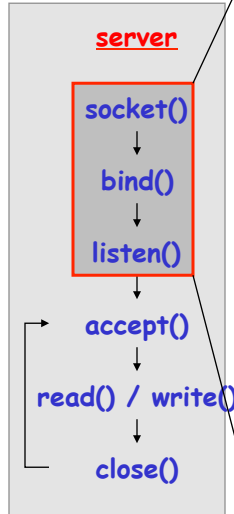
The term **concurrent server** refers to whether the server permits multiple requests to proceed concurrently, **not** to whether the underlying implementation uses multiple, concurrent threads of execution.

Iterative server implementations are easier to build and understand, but may result in poor performance because they make clients wait for service.
- **Connection-Oriented** vs. **Connectionless** Access:
 

Connection-oriented (TCP, typically) servers are easier to implement, but have resources bound to connections. Reliable communication over UDP is not easy!
- **Stateful** vs. **Stateless** Servers:
 

How much information should the server maintain about clients? (What if clients crash, and server does not know?)

### Example: Iterative, Connection-Oriented Server



```

int passiveTCPSock(const char * service, int backlog) {
 struct sockaddr_in sin; /* Internet endpoint address */
 memset(&sin, 0, sizeof(sin)); /* Zero out address */
 sin.sin_family = AF_INET;
 sin.sin_addr.s_addr = INADDR_ANY;

 /* Map service name to port number */
 if (struct servent * pse = getservbyname(service, "tcp"))
 sin.sin_port = pse->s_port;
 else if ((sin.sin_port = htons((unsigned short)atoi(service))) == 0)
 errexit("can't get <service> service entry\n", service);

 /* Allocate socket */
 int s = socket(AF_INET, SOCK_STREAM, 0);
 if (s < 0) errexit("can't create socket: %s\n", strerror(errno));

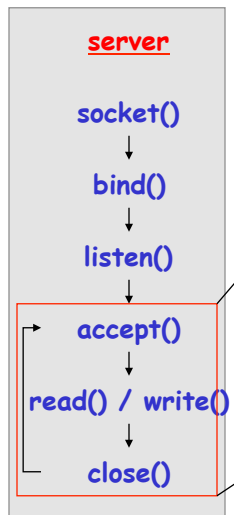
 /* Bind the socket */
 if (bind(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
 errexit("can't bind to ...n");

 /* Listen on socket */
 if (listen(s, backlog) < 0)
 errexit("can't listen on ...n");

 return s;
}

```

### Example: Iterative, Connection-Oriented Server



```

int main(int argc, char * argv[]) {
 char * service = "daytime"; /* service name or port number */
 int m_sock, s_sock; /* master and slave socket */
 service = argv[1];

 int m_sock = passiveTCPSock(service, 32);

 for (;;) {
 s_sock = accept(m_sock, (struct sockaddr*)&fsin, sizeof(fsin));
 if (s_sock < 0) errexit("accept failed: %s\n", strerror(errno));

 time_t now;
 time(&now);
 char * pts = ctime(&now);
 write(s_sock, pts, strlen(pts));

 close(s_sock);
 }
}

```

### Example: Concurrent, Connection-Oriented Server

```

int passiveTCPSock(const char * service, int backlog);

int main(int argc, char * argv[]) {
 char * service = "daytime"; /* service name or port number */
 int m_sock, s_sock; /* master and slave socket */
 service = argv[1];

 int m_sock = passiveTCPSock(service, 32);

 for (;;) {
 s_sock = accept(m_sock, (struct sockaddr*)&fsin, sizeof(fsin));
 if (s_sock < 0) errexit("accept failed: %s\n", strerror(errno));

 if (fork() == 0) { /* child */
 close(m_sock);
 /* handle request here . . . */
 exit(error_code);
 }
 close(s_sock);
 }
}

```

### Example: Concurrent, Connection-Oriented Server

```

int passiveTCPSock(const char * service, int backlog);

int main(int argc, char * argv[]) {
 char * service = "daytime"; /* service name or port number */
 int m_sock, s_sock; /* master and slave socket */
 service = argv[1];

 int m_sock = passiveTCPSock(service, 32);

 signal(SIGCHLD, cleanly_terminate_child);

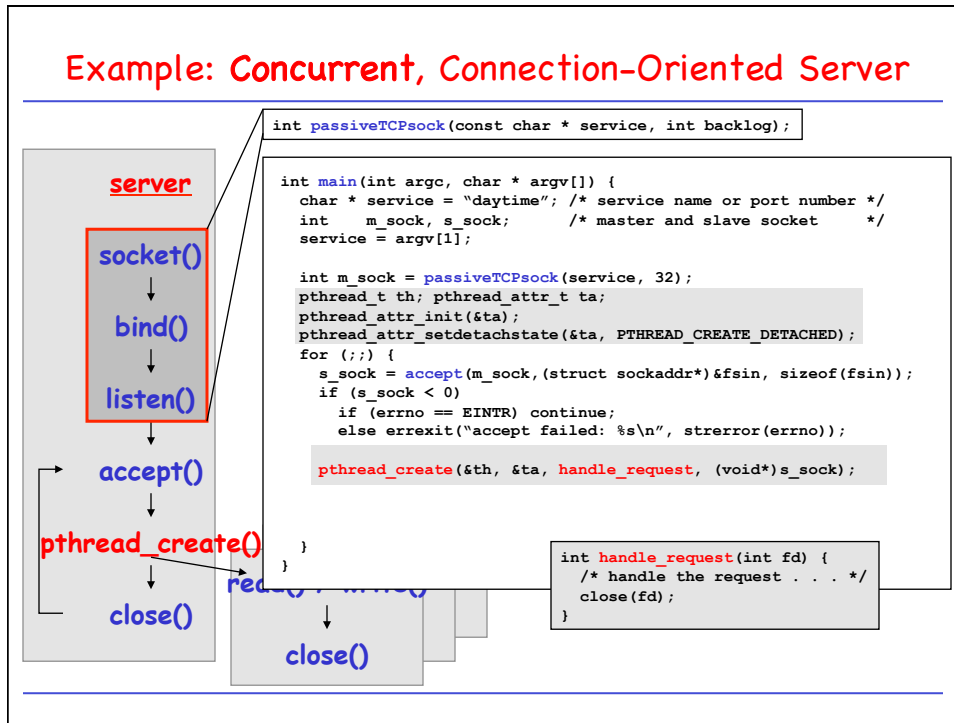
 for (;;) {
 s_sock = accept(m_sock, (struct sockaddr*)&fsin, sizeof(fsin));
 if (s_sock < 0)
 if (errno == EINTR) continue;
 else errexit("accept failed: %s\n", strerror(errno));
 if (fork() == 0) { /* child */
 close(m_sock);
 /* handle request here . . . */
 }
 close(s_sock);
 }
}

void cleanly_terminate_child(int sig) {
 int status;
 while (wait3(&status, WNOHANG, NULL) > 0)
 }

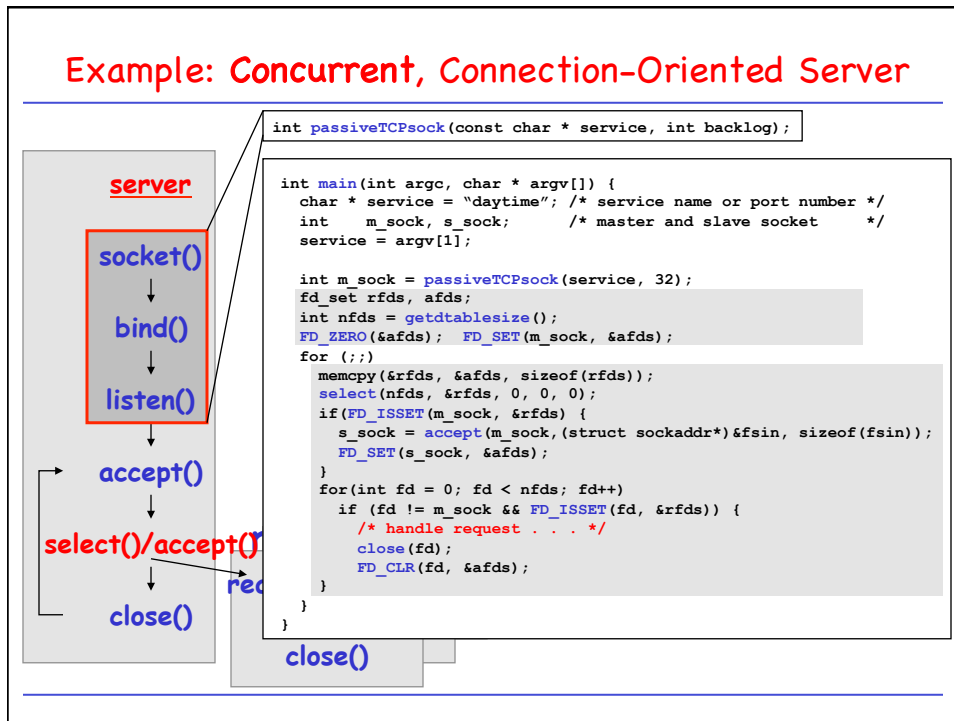
```



### Example: Concurrent, Connection-Oriented Server



### Example: Concurrent, Connection-Oriented Server



## Threaded vs. Event-Driven Design

Figures from: M. Welsh, D. Culler, and E. Brewer, **SEDA: An Architecture for Well Conditioned, Scalable Internet Services**

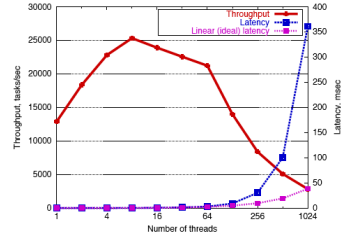


Figure 2: Threaded server throughput degradation: This benchmark measures a simple threaded server which creates a single thread for each task in the pipeline. After receiving a task, each thread performs an 8 KB read from a disk file; all threads read from the same file, so the data is always in the buffer cache.

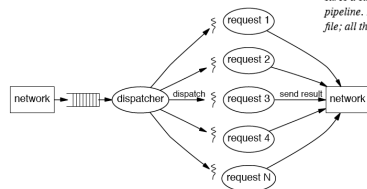


Figure 1: Threaded server design: Each incoming request is dispatched to a separate thread, which processes the request and returns a result to the client. Edges represent control flow between components. Note that other IO operations, such as disk access, are not shown here, but would be incorporated into each threads' request processing.

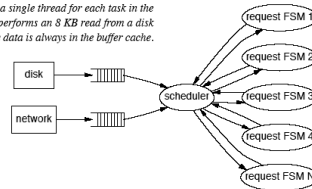


Figure 3: Event-driven server design: This figure shows the flow of events through an event-driven server. The main thread processes incoming events from the network, disk, and other sources, and uses these to drive the execution of many finite state machines. Each FSM represents a single request or flow of execution through the system. The key source of complexity in this design is the event scheduler, which must control the execution of each FSM.

## Why Threads Are A Bad Idea (for most purposes)

**John Ousterhout**

**Sun Microsystems Laboratories**

john.ousterhout@eng.sun.com  
http://www.sunlabs.com/~ouster

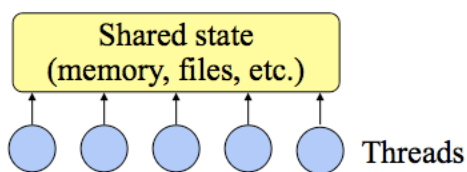
## Introduction

- ◆ **Threads:**
  - Grew up in OS world (processes).
  - Evolved into user-level tool.
  - Proposed as solution for a variety of problems.
  - Every programmer should be a threads programmer?
- ◆ **Problem: threads are very hard to program.**
- ◆ **Alternative: events.**
- ◆ **Claims:**
  - For most purposes proposed for threads, events are better.
  - Threads should be used only when true CPU concurrency is needed.

*Why Threads Are A Bad Idea*

*September 28, 1995, slide 2*

## What Are Threads?



- ◆ **General-purpose solution for managing concurrency.**
- ◆ **Multiple independent execution streams.**
- ◆ **Shared state.**
- ◆ **Pre-emptive scheduling.**
- ◆ **Synchronization (e.g. locks, conditions).**

*Why Threads Are A Bad Idea*

*September 28, 1995, slide 3*

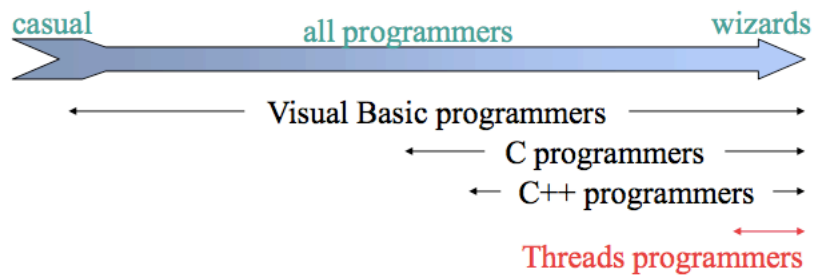
## What Are Threads Used For?

- ◆ **Operating systems:** one kernel thread for each user process.
- ◆ **Scientific applications:** one thread per CPU (solve problems more quickly).
- ◆ **Distributed systems:** process requests concurrently (overlap I/Os).
- ◆ **GUIs:**
  - Threads correspond to user actions; can service display during long-running computations.
  - Multimedia, animations.

*Why Threads Are A Bad Idea*

*September 28, 1995, slide 4*

## What's Wrong With Threads?



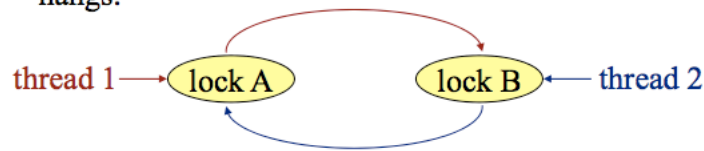
- ◆ **Too hard for most programmers to use.**
- ◆ **Even for experts, development is painful.**

*Why Threads Are A Bad Idea*

*September 28, 1995, slide 5*

## Why Threads Are Hard

- ◆ **Synchronization:**
  - Must coordinate access to shared data with locks.
  - Forget a lock? Corrupted data.
- ◆ **Deadlock:**
  - Circular dependencies among locks.
  - Each process waits for some other process: system hangs.

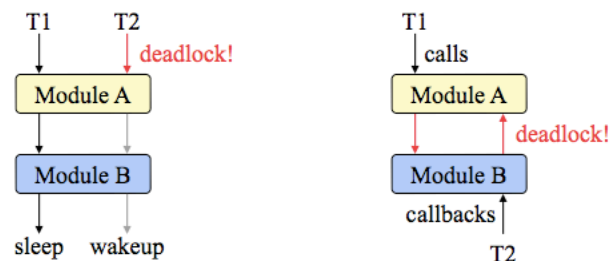


*Why Threads Are A Bad Idea*

*September 28, 1995, slide 6*

## Why Threads Are Hard, cont'd

- ◆ **Hard to debug:** data dependencies, timing dependencies.
- ◆ **Threads break abstraction:** can't design modules independently.
- ◆ **Callbacks don't work with locks.**



*Why Threads Are A Bad Idea*

*September 28, 1995, slide 7*

## Why Threads Are Hard, cont'd

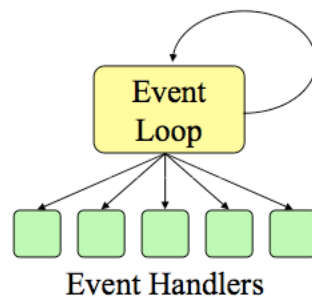
- ◆ **Achieving good performance is hard:**
  - Simple locking (e.g. monitors) yields low concurrency.
  - Fine-grain locking increases complexity, reduces performance in normal case.
  - OSes limit performance (scheduling, context switches).
- ◆ **Threads not well supported:**
  - Hard to port threaded code (PCs? Macs?).
  - Standard libraries not thread-safe.
  - Kernel calls, window systems not multi-threaded.
  - Few debugging tools (LockLint, debuggers?).
- ◆ **Often don't want concurrency anyway (e.g. window events).**

*Why Threads Are A Bad Idea*

*September 28, 1995, slide 8*

## Event-Driven Programming

- ◆ **One execution stream: no CPU concurrency.**
- ◆ **Register interest in events (callbacks).**
- ◆ **Event loop waits for events, invokes handlers.**
- ◆ **No preemption of event handlers.**
- ◆ **Handlers generally short-lived.**



*Why Threads Are A Bad Idea*

*September 28, 1995, slide 9*

## What Are Events Used For?

### ◆ Mostly GUIs:

- One handler for each event (press button, invoke menu entry, etc.).
- Handler implements behavior (undo, delete file, etc.).

### ◆ Distributed systems:

- One handler for each source of input (socket, etc.).
- Handler processes incoming request, sends response.
- Event-driven I/O for I/O overlap.

*Why Threads Are A Bad Idea*

*September 28, 1995, slide 10*

## Problems With Events

### ◆ Long-running handlers make application non-responsive.

- Fork off subprocesses for long-running things (e.g. multimedia), use events to find out when done.
- Break up handlers (e.g. event-driven I/O).
- Periodically call event loop in handler (reentrancy adds complexity).

### ◆ Can't maintain local state across events (handler must return).

### ◆ No CPU concurrency (not suitable for scientific apps).

### ◆ Event-driven I/O not always well supported (e.g. poor write buffering).

*Why Threads Are A Bad Idea*

*September 28, 1995, slide 11*

## Events vs. Threads

- ◆ **Events avoid concurrency as much as possible, threads embrace:**

- Easy to get started with events: no concurrency, no preemption, no synchronization, no deadlock.
- Use complicated techniques only for unusual cases.
- With threads, even the simplest application faces the full complexity.

- ◆ **Debugging easier with events:**

- Timing dependencies only related to events, not to internal scheduling.
- Problems easier to track down: slow response to button vs. corrupted memory.

*Why Threads Are A Bad Idea*

*September 28, 1995, slide 12*

## Events vs. Threads, cont'd

- ◆ **Events faster than threads on single CPU:**

- No locking overheads.
- No context switching.

- ◆ **Events more portable than threads.**

- ◆ **Threads provide true concurrency:**

- Can have long-running stateful handlers without freezes.
- Scalable performance on multiple CPUs.

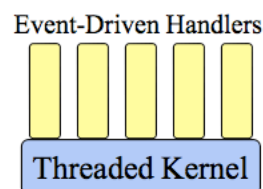
*Why Threads Are A Bad Idea*

*September 28, 1995, slide 13*



## Should You Abandon Threads?

- ◆ **No: important for high-end servers (e.g. databases).**
- ◆ **But, avoid threads wherever possible:**
  - Use events, not threads, for GUIs, distributed systems, low-end servers.
  - Only use threads where true CPU concurrency is needed.
  - Where threads needed, isolate usage in threaded application kernel: keep most of code single-threaded.



*Why Threads Are A Bad Idea*

*September 28, 1995, slide 14*

## Conclusions

- ◆ **Concurrency is fundamentally hard; avoid whenever possible.**
- ◆ **Threads more powerful than events, but power is rarely needed.**
- ◆ **Threads much harder to program than events; for experts only.**
- ◆ **Use events as primary development tool (both GUIs and distributed systems).**
- ◆ **Use threads only for performance-critical kernels.**

*Why Threads Are A Bad Idea*

*September 28, 1995, slide 15*

## A Dissenting Opinion (selected slides)

# Why Events Are A Bad Idea

(for high-concurrency servers)

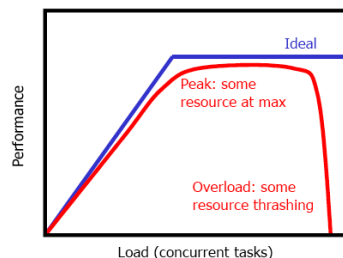


Rob von Behren, Jeremy Condit and Eric Brewer  
University of California at Berkeley  
{jrvb,jcondit,brewer}@cs.berkeley.edu  
<http://capriccio.cs.berkeley.edu>

A Talk HotOS 2003

## The Stage

- Highly concurrent applications
  - Internet servers (Flash, Ninja, SEDA)
  - Transaction processing databases
- Workload
  - Operate "near the knee"
  - Avoid thrashing!
- What makes concurrency hard?
  - Race conditions
  - Scalability (no  $O(n)$  operations)
  - Scheduling & resource sensitivity
  - Inevitable overload
  - Code complexity



## The Debate

- Performance vs. Programmability
  - Current threads pick one
  - Events somewhat better
- Questions
  - Threads vs. Events?
  - How do we get performance and programmability?

*Web Server*

```

graph TD
 A((Accept Conn.)) --> B((Read Request))
 B --> C((Pin Cache))
 C --> D((Write Response))
 D --> E((Exit))
 C --> F((Read File))

```

## The Duality Argument

- General assumption: follow "good practices"
- Observations
  - Major concepts are analogous
  - Program structure is similar
  - Performance should be similar
    - Given good implementations!

| <i>Threads</i>                                                                                                                                                        | <i>Events</i>                                                                                                                                                                   |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>■ Monitors</li> <li>■ Exported functions</li> <li>■ Call/return and fork/join</li> <li>■ Wait on condition variable</li> </ul> | <ul style="list-style-type: none"> <li>■ Event handler &amp; queue</li> <li>■ Events accepted</li> <li>■ Send message / await reply</li> <li>■ Wait for new messages</li> </ul> |

## Our Position

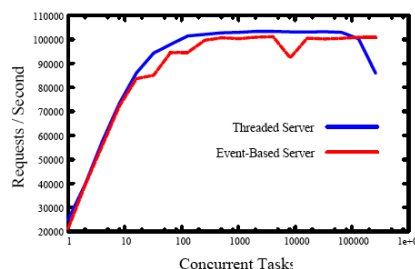
- Thread-event duality still holds
- But threads are better anyway
  - More natural to program
  - Better fit with tools and hardware
- Compiler-runtime integration is key

## “But Events *Are* Better!”

- Recent arguments for events
  - Lower runtime overhead
  - Better live state management
  - Inexpensive synchronization
  - More flexible control flow
  - Better scheduling and locality
- All true but...
  - No *inherent* problem with threads!
  - Thread implementations can be improved

## Runtime Overhead

- *Criticism: Threads don't perform well for high concurrency*
- Response
  - Avoid  $O(n)$  operations
  - Minimize context switch overhead
- Simple scalability test
  - Slightly modified GNU Pth
  - Thread-per-task vs. single thread
  - Same performance!

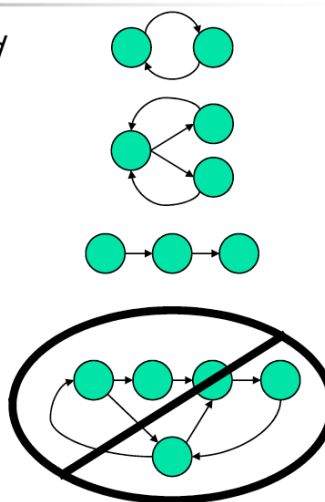


## Synchronization

- *Criticism: Thread synchronization is heavyweight*
- Response
  - Cooperative multitasking works for threads, too!
  - Also presents same problems
    - Starvation & fairness
    - Multiprocessors
    - Unexpected blocking (page faults, etc.)
  - Compiler support helps

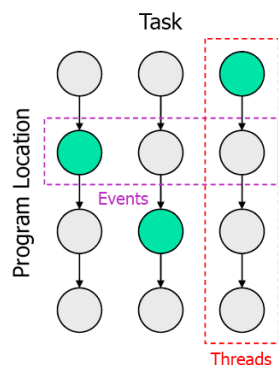
## Control Flow

- *Criticism: Threads have restricted control flow*
- Response
  - Programmers use simple patterns
    - Call / return
    - Parallel calls
    - Pipelines
  - Complicated patterns are unnatural
    - Hard to understand
    - Likely to cause bugs



## Scheduling

- *Criticism: Thread schedulers are too generic*
  - Can't use application-specific information
- Response
  - 2D scheduling: task & program location
    - Threads schedule based on task only
    - Events schedule by location (e.g. SEDA)
      - Allows batching
      - Allows prediction for SRCT
  - Threads can use 2D, too!
    - Runtime system tracks current location
    - Call graph allows prediction

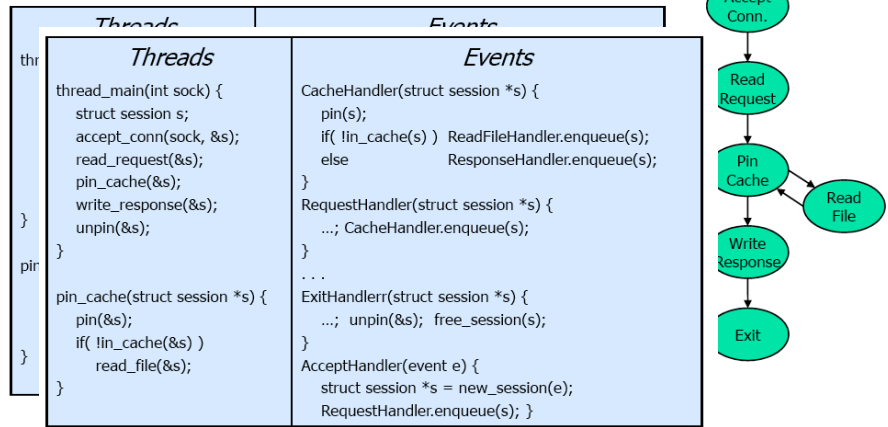


## Our Big But...

- More natural programming model
  - Control flow is more apparent
  - Exception handling is easier
  - State management is automatic
- Better fit with current tools & hardware
  - Better existing infrastructure
  - Allows better performance?

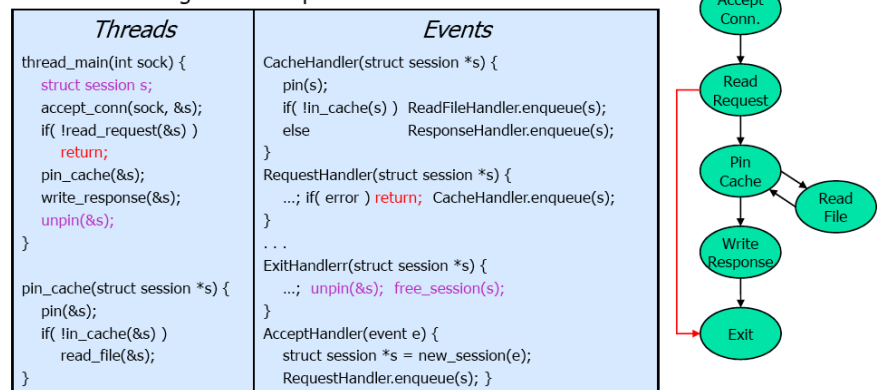
# Control Flow

- Events obscure control flow
  - For programmers *and* tools



# Exceptions

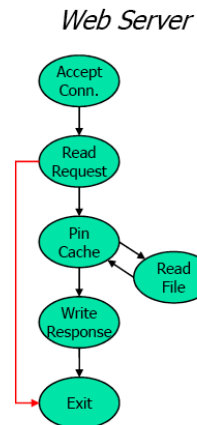
- Exceptions complicate control flow
  - Harder to understand program flow
  - Cause bugs in cleanup code



## State Management

- Events require manual state management
- Hard to know when to free
  - Use GC or risk bugs

| <i>Threads</i>                                                                                                                                                                                                                                                                                                                                  | <i>Events</i>                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> thread_main(int sock) {     struct session s;     accept_conn(sock, &amp;s);     if( !read_request(&amp;s) )         return;     pin_cache(&amp;s);     write_response(&amp;s);     unpin(&amp;s); }  pin_cache(struct session *s) {     pin(&amp;s);     if( !in_cache(&amp;s) )         read_file(&amp;s); }                     </pre> | <pre> CacheHandler(struct session *s) {     pin(s);     if( !in_cache(s) ) ReadFileHandler.enqueue(s);     else ResponseHandler.enqueue(s); }  RequestHandler(struct session *s) {     ...; if( error ) return; CacheHandler.enqueue(s); }  ...  ExitHandler(struct session *s) {     ...; unpin(&amp;s); free_session(s); }  AcceptHandler(event e) {     struct session *s = new_session(e);     RequestHandler.enqueue(s); }                     </pre> |



## Existing Infrastructure

- Lots of infrastructure for threads
  - Debuggers
  - Languages & compilers
- Consequences
  - More amenable to analysis
  - Less effort to get working systems

## Better Performance?

- Function pointers & dynamic dispatch
  - Limit compiler optimizations
  - Hurt branch prediction & I-cache locality
- More context switches with events?
  - Example: Haboob does 6x more than Knot
  - Natural result of queues
- More investigation needed!



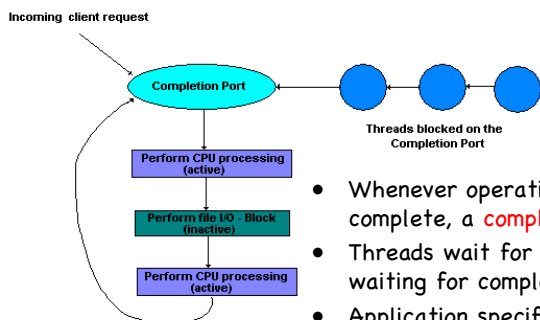
## The Future: Compiler-Runtime Integration

- Insight
  - Automate things event programmers do by hand
  - Additional analysis for other things
- Specific targets
  - Dynamic stack growth\*
  - Live state management
  - Synchronization
  - Scheduling\*
- Improve performance *and* decrease complexity

## Event-Driven Programming in Practice: Completion Ports

- Rationale:
  - Minimize context switches by having threads avoid unnecessary blocking.
  - Maximize parallelism by using multiple threads.
  - Ideally, have one thread actively servicing a request on every processor.
  - Do not block thread if there are additional requests waiting when thread completes a request.
  - The application must be able to activate another thread when current thread blocks on I/O (e.g. when it reads from a file)
- Resources:
  - Inside IO Completion Ports:  
<http://technet.microsoft.com/en-us/sysinternals/bb963891.aspx>
  - Multithreaded Asynchronous I/O & I/O Completion Ports:  
<http://www.ddj.com/cpp/20120292>
  - Parallel Programming with C++ - I/O Completion Ports:  
<http://weblogs.asp.net/kennykerr/archive/2008/01/03/parallel-programming-with-c-part-4-i-o-completion-ports.aspx>

## Completion Ports (CPs): Operation



- Whenever operations on files associated with CP's complete, a **completion packet** is queued on the CP.
- Threads wait for outstanding I/Os to complete by waiting for completion packets to be queued on CP.
- Application specifies **concurrency value** associated with CP.
- Whenever active thread finishes processing current request, it checks for next packet at the port. (If there is, grabs it without context switch.)
- Whenever a thread gets **blocked**, the number of active threads drops below concurrency value, and next thread can start.

## Basic Steps for Using Completion Ports

1. Create a new I/O completion port object.
2. Associate one or more file descriptors with the port.
3. Issue asynchronous read/write operations on the file descriptor(s).
4. Retrieve completion notifications from the port and handle accordingly.

Multiple threads may monitor a single I/O completion port and retrieve completion events—the operating system effectively manages the thread pool, ensuring that the completion events are distributed efficiently across threads in the pool.

## Completion Ports: APIs:

---

### CP creation and association of file descriptor with CP:

```
HANDLE CreateIoCompletionPort(
 HANDLE FileHandle, /* INVALID... when creating new CP*/
 HANDLE ExistingCompletionPort, /* NULL when creating new CP */
 DWORD CompletionKey, /* NULL when creating new CP */
 DWORD NumberOfConcurrentThreads /* Concurrency value */
);
```

### Initiating Asynchronous I/O Request:

```
BOOL ReadFile(
 HANDLE FileHandle,
 LPVOID pBuffer,
 DWORD NumberOfBytesToRead,
 LPDWORD pNumberOfBytesRead,
 LPOVERLAPPED pOverlapped /* specify parameters
 and receive results */
);
```

---

## Completion Ports: APIs (Remove and Post CP Events)

---

### Retrieve next completion packet:

```
BOOL GetQueuedCompletionStatus(
 HANDLE CompletionPort,
 LPDWORD lpNumberOfBytesTransferred,
 LPDWORD CompletionKey,
 LPOVERLAPPED* ppOverlapped, /* pointer to pointer parameter to
 asynch I/O function */
 DWORD dwMillisecondTimeout
);
```

### Generate completion packets (send implementation-specific events):

```
BOOL PostQueuedCompletionStatus(
 HANDLE CompletionPort,
 LPDWORD lpNumberOfBytesTransferred,
 LPDWORD CompletionKey,
 LPOVERLAPPED lpOverlapped
);
```

When CP event gets posted on a CP, one of the waiting threads returns from call to **GetQueuedCompletionStatus** with copies of parameters as they were posted.

---

## CP Example: Web Server: Startup

Tom R. Dial, "Multithreaded Asynchronous I/O & I/O Completion Ports," Dr. Dobbs, Aug.2007)

```

/* Fire.cpp - The Fire Web Server
 * Copyright (C) 2007 Tom R. Dial tdial@kavaga.com */
int main(int /*argc*/, char* /*argv*/[]) {
 // Initialize the Microsoft Windows Sockets Library
 WSADATA Wsa={0};
 WSASStartup(MAKEWORD(2,2), &Wsa);
 // Get the working directory; this is used when transmitting files back.
 GetCurrentDirectory(_MAX_PATH, RootDirectory);
 // Create an event to use to synchronize the shutdown process.
 StopEvent = CreateEvent(0, FALSE, FALSE, 0);
 // Setup a console control handler: We stop the server on CTRL-C
 SetConsoleCtrlHandler(ConsoleCtrlHandler, TRUE);

 // Create a new I/O Completion port.
 HANDLE IoPort = CreateIoCompletionPort(INVALID_HANDLE_VALUE, 0, 0, WORKER_THREAD_COUNT);

 // Set up a socket on which to listen for new connections.
 SOCKET Listener = WSASocket(PF_INET, SOCK_STREAM, IPPROTO_TCP, 0, 0, WSA_FLAG_OVERLAPPED);
 struct sockaddr_in Addr={0};
 Addr.sin_family = AF_INET;
 Addr.sin_addr.S_un.S_addr = INADDR_ANY;
 Addr.sin_port = htons(DEFAULT_PORT);
 // Bind the listener to the local interface and set to listening state.
 bind(Listener, (struct sockaddr*)&Addr, sizeof(struct sockaddr_in));
 listen(Listener, DEFAULT_LISTEN_QUEUE_SIZE);

```

## CP Example: Web Server: Start Threads

```

// Create worker threads
HANDLE Workers[WORKER_THREAD_COUNT] = 0;
unsigned int WorkerIds[WORKER_THREAD_COUNT] = 0 ;

for (size_t i=0; i<WORKER_THREAD_COUNT; i++)
 Workers[i] = (HANDLE)_beginthreadex(0, 0, WorkerProc, IoPort, 0, WorkerIds+i);

// Associate the Listener socket with the I/O Completion Port.
CreateIoCompletionPort((HANDLE)Listener, IoPort, COMPLETION_KEY_IO, 0);

// Allocate an array of connections; constructor binds them to the port.
Connection* Connections[MAX_CONCURRENT_CONNECTIONS]={0};
for (size_t i=0; i<MAX_CONCURRENT_CONNECTIONS; i++)
 Connections[i] = new Connection(Listener, IoPort);

// Print instructions for stopping the server.
printf("Fire Web Server: Press CTRL-C To shut down.\n");
// Wait for the user to press CTRL-C...
WaitForSingleObject(StopEvent, INFINITE);

// ...

```

## CP Example: Web Server: Shutdown

```
// Deregister console control handler: We stop the server on CTRL-C
SetConsoleCtrlHandler(NULL, FALSE);
// Post a quit completion message, one per worker thread.
for (size_t i=0; i<WORKER_THREAD_COUNT; i++)
 PostQueuedCompletionStatus(IoPort, 0, COMPLETION_KEY_SHUTDOWN, 0);
// Wait for all of the worker threads to terminate...
WaitForMultipleObjects(WORKER_THREAD_COUNT, Workers, TRUE, INFINITE);
// Close worker thread handles.
for (size_t i=0; i<WORKER_THREAD_COUNT; i++)
 CloseHandle(Workers[i]);
// Close stop event.
CloseHandle(StopEvent);
// Shut down the listener socket and close the I/O port.
shutdown(Listener, SD_BOTH);
closesocket(Listener);
CloseHandle(IoPort);
// Delete connections.
for (size_t i=0; i<MAX_CONCURRENT_CONNECTIONS; i++)
 delete(Connections[i]);
WSACleanup();
return 0;
}
```

## CP Example: Web Server: Worker Threads

```
// Worker thread procedure.
unsigned int __stdcall WorkerProc(void* IoPort) {
 for (;;) {
 BOOL Status = 0;
 DWORD NumTransferred = 0;
 ULONG_PTR CompKey = COMPLETION_KEY_NONE;
 LPOVERLAPPED pOver = 0;
 Status = GetQueuedCompletionStatus(reinterpret_cast<HANDLE>(IoPort),
 &NumTransferred, &CompKey, &pOver, INFINITE);
 Connection* pConn = reinterpret_cast<Connection*>(pOver);
 if (FALSE == Status) {
 // An error occurred; reset to a known state.
 if (pConn) pConn->IssueReset();
 } else if (COMPLETION_KEY_IO == CompKey) {
 pConn->OnIoComplete(NumTransferred);
 } else if (COMPLETION_KEY_SHUTDOWN == CompKey) {
 break;
 }
 }
 return 0;
}
```

## CP Example: Web Server: Connections

```
// Class representing a single connection.

class Connection : public OVERLAPPED {
 enum STATE { WAIT_ACCEPT = 0, WAIT_REQUEST = 1,
 WAIT_TRANSMIT = 2, WAIT_RESET = 3 };
public:
 Connection(SOCKET Listener, HANDLE IoPort) : myListener(Listener) {
 myState = WAIT_ACCEPT;
 // [...]
 mySock = WSASocket(PF_INET, SOCK_STREAM, IPPROTO_TCP,
 0, 0, WSA_FLAG_OVERLAPPED);
 // Associate the client socket with the I/O Completion Port.
 CreateIoCompletionPort(reinterpret_cast<HANDLE>(mySock),
 IoPort, COMPLETION_KEY_IO, 0);
 IssueAccept();
 }
 ~Connection() {
 shutdown(mySock, SD_BOTH);
 closesocket(mySock);
 }
}
```

## CP Example: Web Server: State Machines (I)

```
// ACCEPT OPERATION

// Issue an asynchronous accept.

void Connection::IssueAccept() {
 myState = WAIT_ACCEPT;
 DWORD ReceiveLen = 0; // This gets thrown away, but must be passed.
 AcceptEx(myListener, mySock, myAddrBlock, 0, ACCEPT_ADDRESS_LENGTH,
 ACCEPT_ADDRESS_LENGTH, &ReceiveLen, (OVERLAPPED*)this);
}

// Complete the accept and update the client socket's context.

void Connection::CompleteAccept() {
 setsockopt(mySock, SOL_SOCKET, SO_UPDATE_ACCEPT_CONTEXT,
 (char*)&myListener, sizeof(SOCKET));
 // Transition to "reading request" state.
 IssueRead();
}
```

## CP Example: Web Server: State Machines (II)

```

// READ OPERATION

// Issue an asynchronous read operation.
void Connection::IssueRead(void) {
 myState = WAIT_REQUEST;
 ReadFile((HANDLE)mySock, myReadBuf, DEFAULT_READ_BUFFER_SIZE,
 0, (OVERLAPPED*)this);
}

// Complete the read operation, appending the request with the latest data.
void Connection::CompleteRead(size_t NumBytesRead) {
 // [...]
 // Has the client finished sending the request?
 if (IsRequestComplete(NumBytesRead)) {
 // Yes. Transmit the response.
 IssueTransmit();
 } else {
 // The client is not finished. If data was read this pass, we assume the connection
 // is still good and read more. If not, we assume that the client closed the socket
 // prematurely.
 if (NumBytesRead) IssueRead();
 else IssueReset();
 }
}

```

## CP Example: Web Server: State Machines (III)

```

// Parse the request, and transmit the response.
void Connection::IssueTransmit() {
 myState = WAIT_TRANSMIT;
 // Simplified parsing of the request: just ignore first token.
 char* Method = strtok(&myRequest[0], " ");
 if (!Method) {
 IssueReset();
 return;
 }
 // Parse second token, create file, transmit file ..
 // [...]
 myFile = CreateFile(/* ... */);
 TransmitFile(mySock, myFile,
 Info.nFileSizeLow, 0, this,
 &myTransmitBuffers, 0);
}

void Connection::CompleteTransmit() {
 // Issue the reset; this prepares the
 // socket for reuse.
 IssueReset();
}

void Connection::IssueReset()
{
 myState = WAIT_RESET;
 TransmitFile(mySock, 0, 0, 0, this, 0,
 TF_DISCONNECT | TF_REUSE_SOCKET);
}

void Connection::CompleteReset(void)
{
 ClearBuffers();
 IssueAccept(); // Continue to next request!
}

```

## CP Example: Web Server: Dispatching

```
// The main handler for the connection, responsible for state transitions.

void Connection::OnIoComplete(DWORD NumTransferred) {

 switch (myState) {
 case WAIT_ACCEPT:
 CompleteAccept();
 break;
 case WAIT_REQUEST:
 CompleteRead(NumTransferred);
 break;
 case WAIT_TRANSMIT:
 CompleteTransmit();
 break;
 case WAIT_RESET:
 CompleteReset();
 break;
 }

}
```