Event-Based Programming
(us concurrent programs don't need no threads!)

- Event-driven programming old-style: UNIX select/accept.
- Event-driven programming newer-style: Windows Completion Ports.

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.

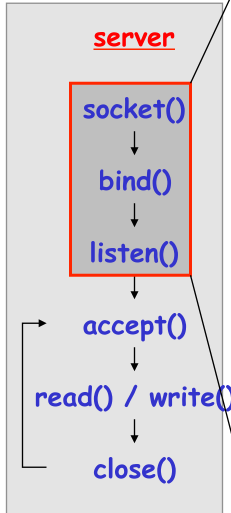
  Iterative server implementations are easier to build and understand, but may result in poor performance because they make clients wait for service.
- Connection-Oriented vs. Connectionless Access:

  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:

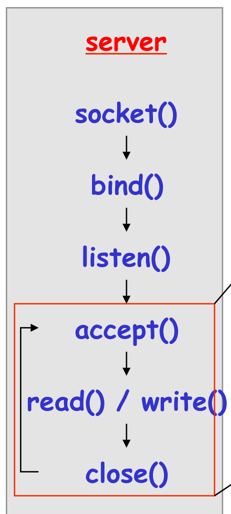
  How much information should the server maintain about clients? (What if clients crash, and server does not know?)

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 <%s> 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

server

socket() → bind() → listen() → accept() → fork() → close()

rec... → close()

```
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

server

socket() → bind() → listen() → accept() → fork() → close()

read() / write() → close()

```
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


```
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);
  pthread_t th; pthread_attr_t ta;
  pthread_attr_init(&ta);
  pthread_attr_setdetachstate(&ta, PTHREAD_CREATE_DETACHED);
  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));

    pthread_create(&th, &ta, handle_request, (void*)s_sock);

  }
}
```

```
int handle_request(int fd) {
  /* handle the request . . . */
  close(fd);
}
```

Example: **Concurrent**, Connection-Oriented Server

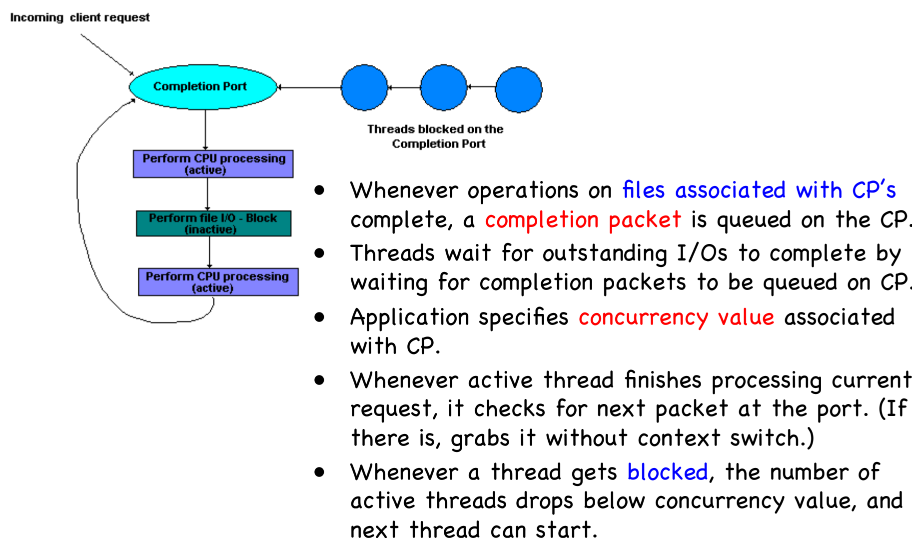

```
int passiveTCPsock(const char * service, int backlog);
```

```
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);
  fd_set rfds, afds;
  int nfds = getdtablesize();
  FD_ZERO(&afds);  FD_SET(m_sock, &afds);
  for (;;)
    memcpy(&rfds, &afds, sizeof(rfds));
    select(nfds, &rfds, 0, 0, 0);
    if(FD_ISSET(m_sock, &rfds) {
      s_sock = accept(m_sock,(struct sockaddr*)&fsin, sizeof(fsin));
      FD_SET(s_sock, &afds);
    }
    for(int fd = 0; fd < nfds; fd++)
      if (fd != m_sock && FD_ISSET(fd, &rfds)) {
        /* handle request . . . */
        close(fd);
        FD_CLR(fd, &afds);
    }
  }
}
```

Event-Driven Programming in Practice: Windows 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};
   WSAStartup( 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: 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;
    }

}
```

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!
  }
```