Threading, Events, and Concurrency

- Threading Recap
- Threading in Multicore World
- User-Level Threads vs. Kernel-Level Threads
  - Example: Scheduler Activations
- Thread-based vs. Event-based Concurrency
  - Example: Windows Fibers

History

- 1960's
  - First "multiprocessors"
- 1980's
  - Multiprocessing grows, primarily in academia and other research settings.
- 1990's
  - Multiprocessors become widely available in the market place.
  - Symmetric multiprocessing requires changes to OSs
  - "Memory wall"
- More recently:
  - …

Concurrency and Performance: the "Why?"

Latency Reduction:

- Apply parallel algorithm.
- Concurrency in trivially parallelizable problems.

Latency Hiding:

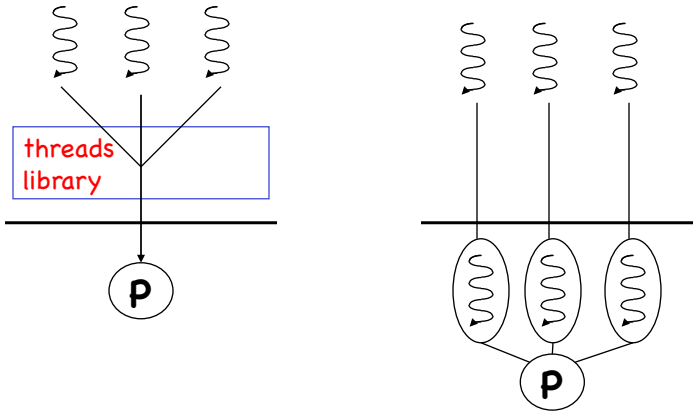
- Use concurrency to perform useful work while another operation is pending.
- Latency of operation is not affected, but hidden.
- Alternatives to concurrent execution:
  - Non-blocking operations (asynchronous I/O)
  - Event loops (`poll()`/`select()`, or completion ports)

Throughput Increase:

- Employ multiple concurrent executions of sequential threads to accommodate more simultaneous work.
- Concurrency is then handled by specialized subsystems (OS, database, etc.)

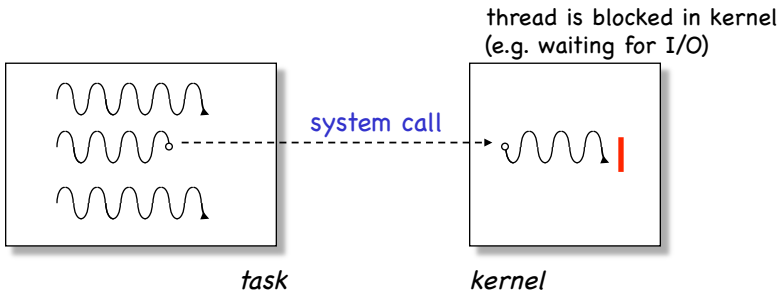
Threads Recap: User vs. Kernel-Level Threads

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

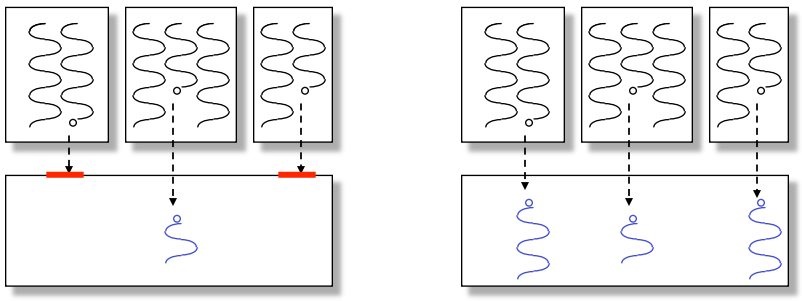


Threads Recap: Potential Problems with Threads

- 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.



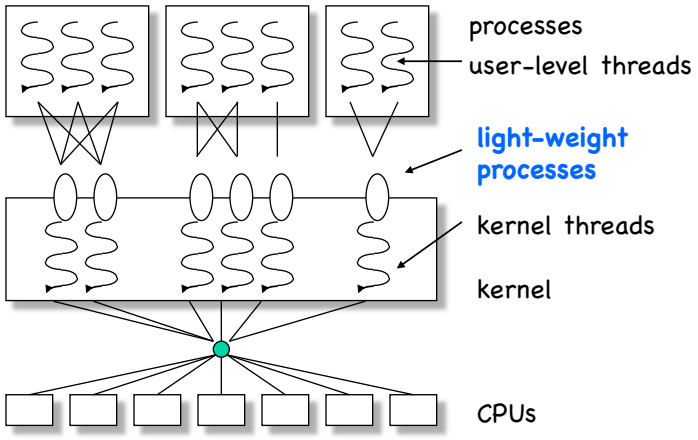
Threads Recap: Singlethreaded vs. Multithreaded **Kernel**



- 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 Recap: Hybrid Multithreading



Threading, Events, and Concurrency

- Threading Recap
- Threading in Multicore World
- User-Level Threads vs. Kernel-Level Threads
  - Example: Scheduler Activations
- Thread-based vs. Event-based Concurrency
  - Example: Windows Fibers

User- vs. Kernel-Level Threads: Scheduler Activations

Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy, "Scheduler Activations: Effective Kernel Support for the User-level Management of Parallelism". ACM SIGOPS Operating Systems Review, Volume 25, Issue 5, Oct. 1991.

User- vs. Kernel-Level Threads

User-Level Threads:

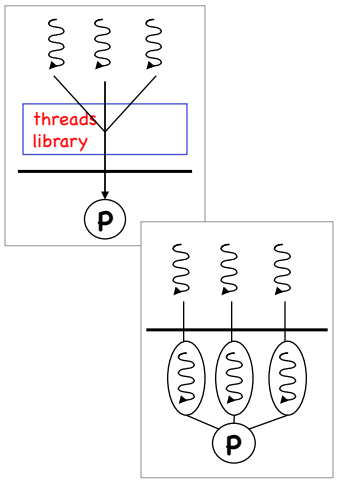
- Managed by runtime library.
- Management operations require no kernel intervention.
- (+) Low-cost
- (+) Flexible (various APIs: POSIX, Actors, …)
- (+) Implementation requires no change to OS.
- (-) Performance issues due to mapping to OS resources (see later)

Kernel-Level Threads:

- (+) Avoid system integration problems (see later)
- (-) Too heavyweight
- -> "user-level threads have ultimatively been implemented on top of the kernel threads of both Mach and Topaz"

"Dilemma":

- "employ kernel threads, which 'work right' but perform poorly, or employ user-level threads implemented on top of kernel threads or processes, which perform well but are functionally deficient."



Goals of Scheduler Activations

- **Functionality:**
  - Should mimic behavior of kernel thread management system:
    - No idling processor in presence of ready threads.
    - No priority inversion
    - Multiprogramming within and across address spaces
- **Performance:**
  - Keep thread management overhead to same as user-level threads.
- **Flexibility:**
  - Allow for changes in scheduling policies or even different concurrency models (workers, Actors, Futures).

User-Level Threads: Advantages

Kernel-level threads have inherent disadvantages

- **Cost of accessing thread management operations**: Must cross protection boundary on every thread operation, even for operations on threads of the same address space
- **Cost of generality:** A single implementation must be used by all applications.
  - In contrast, user-level libraries can be tuned to applications.

Operation	FastThreads	Topaz threads	Ultrix processes
Null Fork	34	948	11300
Signal-Wait	37	441	1840

Table 1: Thread Operation Latencies (μsec.) This data is old!!

User-Level Threads: Limitations

It has been difficult to implement user-level threads and integrate them with system services, because

"Kernel threads are the **wrong abstraction** for supporting user-level thread management":

1. Kernel events, such as processor preemption and I/O blocking and resumption, are handled by the kernel invisibly to the user level.
2. Kernel threads are scheduled obliviously with respect to the user-level thread state.

Scenario: "When a user-level thread makes a blocking I/O request or takes a page fault, the kernel thread serving as its virtual processor also blocks. As a result, the physical processor is lost to the address space while the I/O is pending, …"

User-Level Threads: Limitations (cont)

Scenario: "When a user-level thread makes a blocking I/O request or takes a page fault, the kernel thread serving as its virtual processor also blocks. As a result, the physical processor is lost to the address space while the I/O is pending, …"

Solution (?): "create more kernel threads than physical processors; when one kernel thread blocks because its user-level thread blocks in the kernel, another kernel thread is available to run user-level threads on that processor."

However: When the thread unblocks, there will be more runnable kernel threads than processors. -> The OS now decides on behalf of the application which user-level threads to run.

User-Level Threads: Limitations (cont)

However: When the thread unblocks, there will be more runnable kernel threads than processors. -> The OS now decides on behalf of the application which user-level threads to run.

Solution (?) : "… the operating system could employ some kind of time-slicing to ensure each thread makes progress."

However: "When user-level threads are running on top of kernel threads, time-slicing can lead to problems."

"For example, a kernel thread could be preempted while its user-level thread is holding a spin-lock;
any user-level threads accessing the lock will then spin-wait until the lock holder is re-scheduled."

Similar problems occur when handling multiple jobs.

User-Level Threads: Limitations (cont)

Logical correctness of user-level thread system built on kernel threads…

Example: "Many applications, particularly those that require coordination among multiple address spaces, are free from deadlock based on the assumption that all runnable threads eventually receive processor time."

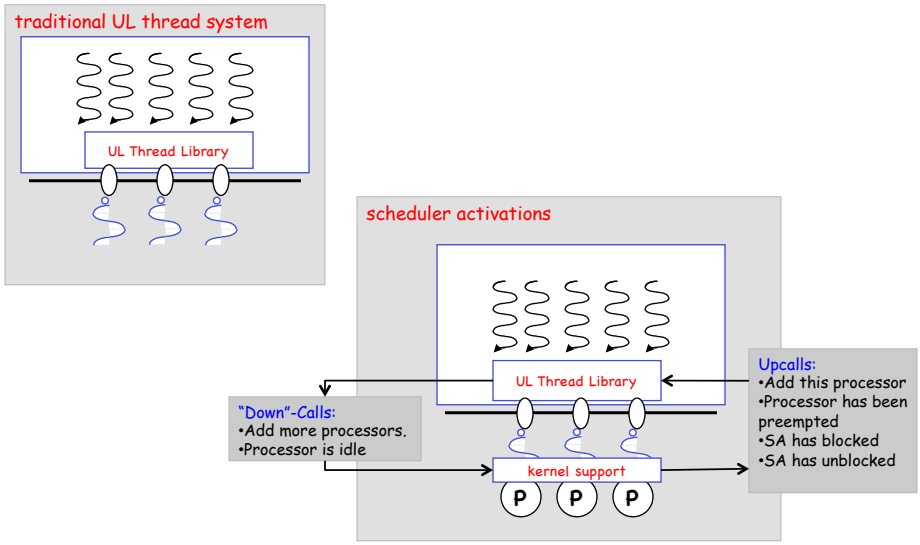
However: "But when user-level threads are multiplexed across a fixed number of kernel threads, the assumption may no longer hold:
because a kernel thread blocks when its user-level thread blocks, an application can run out of kernel threads to serve as execution contexts, even when there are runnable user-level threads and available processors."

SOLUTION: Kernel-Level Support for User-level Threads

- User-level thread system + new kernel interface
- "kernel provides each UL thread system with its own virtual multiprocessor"
- "number of processors in that machine may change during the execution of the program"

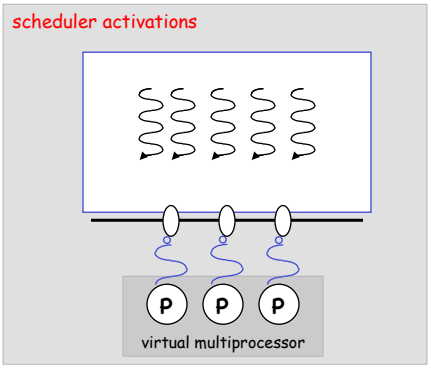
- Abstraction enforces following criteria:
  - Kernel allocates physical processors to address spaces.
  - UL thread system has complete control over which thread to run on allocated processors. (as opposed to earlier limitations)
  - UL thread system is informed whenever number of allocated processors changes.
  - UL thread system knows about suspended/resumed threads in kernel.
  - UL thread system can request/release processors.
  - UL thread system transparent to user. (i.e., user sees KL threads)

Solution: "Scheduler Activations"

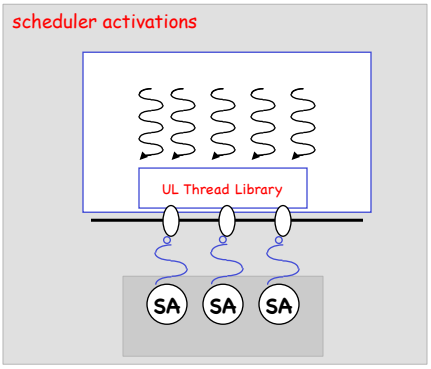


"Scheduler Activations": Abstraction vs. Implementation

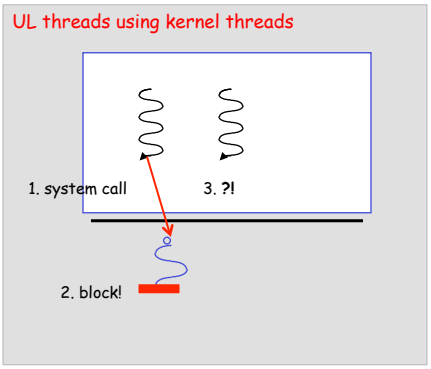
Abstraction:

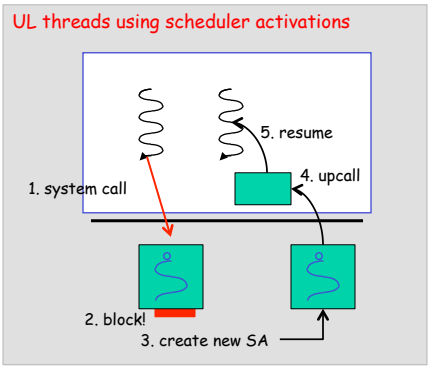


Implementation:

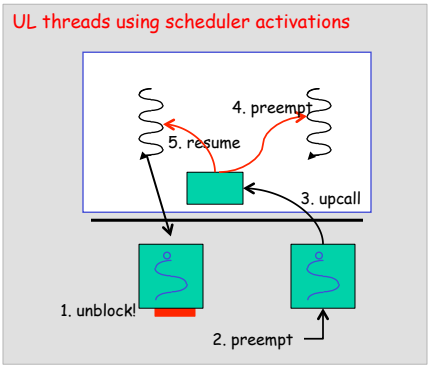


"Scheduler Activations": How to Handle "Blocking" Threads





"Scheduler Activations": Resuming Blocked Threads



Threading, Events, and Concurrency

- Threading Recap
- Threading in Multicore World
- User-Level Threads vs. Kernel-Level Threads
  - Example: Scheduler Activations
- Thread-based vs. Event-based Concurrency
  - Example: Windows Fibers

Recap: Threaded vs. Event-Driven Design

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

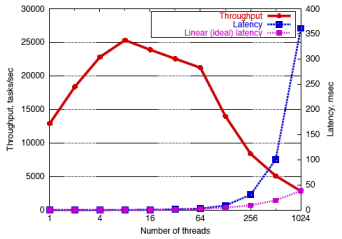


Figure 2: Threaded server throughput degradation: *This benchmark measures a simple threaded server which creates a single thread for each task in the pipeline. After receiving a task, each thread performs an 8 KB read from a disk file; all threads read from the same file, so the data is always in the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

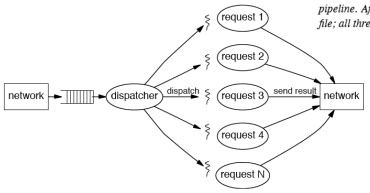


Figure 1: **Threaded server design:** *Each incoming request is dispatched to a separate thread, which processes the request and returns a result to the client. Edges represent control flow between components. Note that other I/O operations, such as disk access, are not shown here, but would be incorporated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

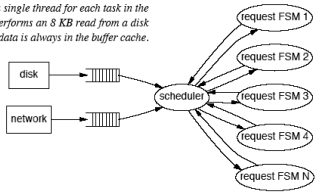


Figure 3: Event-driven server design: *This figure shows the flow of events through an event-driven server. The main thread processes incoming events from the network, disk, and other sources, and uses these to drive the execution of many finite state machines. Each FSM represents a single request or 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.*

Windows Fibers

Aul Adya, Jon Howell, Marvin Theimer, William Bolosky, John R. Douceur, "Cooperative Task Management without Manual Stack Management". Proceedings of the 2002 Usenix Annual Technical Conference, Monterey, CA, June 2002.

Task Management

- **Question**: How do we achieve multiprogramming, concurrency?
- Definition [**Task**]: Control flow. Tasks have access to shared global state.
- **Preemptive** Task Management:
  - Execution of tasks can interleave.
- **Serial** Task Management:
  - Execute each task to completion before starting new task.
- **Cooperative** Task Management:
  - (Voluntarily) yield CPU at well-defined points in execution.

Serial Task Management

Pros:

- Only one task is running at a given time.
- No potential for conflict in accessing shared state.
- We can define so-called "inter-task invariants"; while one task is running, no other task can violate these invariants.

Cons:

- Only one task is running at a given time!
- No multiprogramming.
- No multiprocessor parallelism.

Cooperative Task Management

Pros:

- Allows for some controlled multiprogramming.
- Invariants must be ensured at yielding points only.

Cons:

- Invariants are not automatically enforced.

About those invariants . . .

- We need to ensure that local state does not depend on invalid assumptions about shared state when we resume after yield.
- Example: We want to open file before the yield. Is the file still there after we resume?

Conflict Management

Q: How to avoid inter-task conflicts on shared state?

In serial task management: No problem! Entire task is an atomic operation.

In cooperative task management: Event handlers are basically atomic units of operation.

Conflict Management (2)

Q: How to avoid inter-task conflicts on shared state?

In preemptive task management: Invariants on the shared state must hold all the time. (?!)

- Pessimistic synchronization primitives: Limit the preemptivity to ensure that invariants hold when preemption happen.
- Optimistic synchronization primitives: Speculatively execute, but then roll back if invariants have been violated.

Cooperative Mgmt & Stack Management

Q: How to **realize** cooperative task management?

A solution: **Event Handlers**

Example:
(1) Receive network message
(2) Read block from disk
(3) Reply to message

Event Handlers & Stack Management

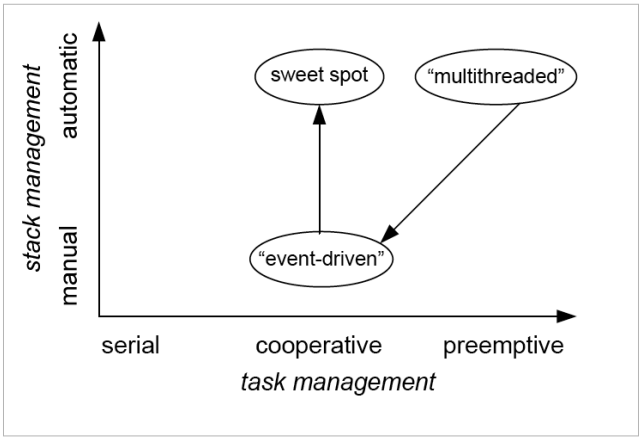
A solution: **Event Handlers**

Example:
(1) Receive network message
(2) Read block from disk
(3) Reply to message

Pros:
- Concurrency

Cons:
- Control flow for single task is broken up across multiple procedures.
- We now have to explicitly carry local state across procedures. (**"Manual Stack Management"**)

Stack Management vs. Task Management



Automatic Stack Management

```
CAInfo GetCAInfo(CAID caId) {
   CAInfo caInfo = LookupHashTable(caId);
   return caInfo;
}
```

in-memory

use on-disk structure (automatic)

```
CAInfo GetCAInfoBlocking(CAID caId) {
   CAInfo caInfo = LookupHashTable(caId);
   if (caInfo != NULL) {
      // Found node in the hash table
      return caInfo;
   }

   caInfo = new CAInfo();
   // DiskRead blocks waiting for
   // the disk I/O to complete.
   DiskRead(caId, caInfo);
   InsertHashTable(caId, CaInfo);
   return caInfo;
}
```

Manual Stack Management

```
class Continuation {
   // The function called when
   // this continuation is
   // scheduled to run.
   void (*function)
          (Continuation cont);
   // Return value set by the
   // I/O operation. To be
   // passed to continuation.
   void *returnValue
   // Bundled up state
   void *arg1, *arg2, ...;
}
```

```
void GetCAInfoHandler1(
                  CAID caId,
                  Continuation *callerCont){
   // Return the result immediately if in cache
   CAInfo *caInfo = LookupHashTable(caId);
   if (caInfo != NULL) {
      // Call caller's continuation with result
      (callerCont.function)(caInfo);
      return;
   }
   // Make buffer space for disk read
   caInfo = new CAInfo();
   // Save return address & live variables
   Continuation *cont =
       new Continuation(&GetCAInfoHandler2,
                        caId, caInfo, callerCont);
   // Send request
   EventHandle eh =
       InitAsyncDiskRead(caId, caInfo);
   // Schedule event handler to run on reply
   // by registering continuation
   RegisterContinuation(eh, cont);
}
```

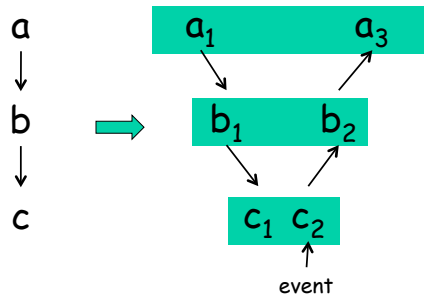
```
void GetCAInfoHandler2(Continuation *cont) {
   // Recover live variables
   CAID caId = (CAID) cont->arg1;
   CAInfo *caInfo = (CAInfo*) cont->arg2;
   Continuation *callerCont =
              (Continuation*) cont->arg3;
   // Stash CAInfo object in hash
   InsertHashTable(caId, caInfo);
   // Now "return" results to original caller
   (callerCont.function)(callerCont);
}
```

Stack Ripping

- Programmer must explicitly save local state and then restore it later.
- Without ripped functions, this would all be managed by the compiler!
- Problems with stack ripping:
  - function scoping: logic is distributed over multiple functions.
  - automatic variables: local state is no more stored on stack.
  - self-propagation of function ripping:
    - A ripped function may require all functions up the call tree to be ripped in two as well. (see figure)
    - Calls to ripped functions in control structures may require complicated ripping of calling function. (see figure)

Stack Ripping

All functions up the calling tree must be ripped.



Stack Ripping in Control Structures

```
<some code here>
while (x < 0)  {
    …
    c();
    …
}
<some more code here>
```

```
<some code here>
while (x < 0)  {
    …
    c1();
```

```
    c2();
    …
    ?!
}
<some more code here>
```

Stack Ripping in Control Structures

```
<some code here>
while (x < 0)  {
    …
    c1();
```

```
    c2();
    …
    ?!
}
<some more cod
```

```
<some code here>
f1();
```

```
function f3() {
    <some more code here>
}
```

```
function f2() {
    c2();
    …
    invoke cont f1;
}
```

```
function f1() {
    if (x < 0)  {
        …
        c1(f2);
    } else {
        invoke cont f3;
}
```

event!

Problems with Concurrency Assumptions

Q: What if a non-yielding function is re-implemented to become yielding?

Q: What are the implications for calling functions?

Question: Would we even know?!

Is this a problem for manual stack management?

How about automatic stack management?

Solutions?!

- Tools!

Static check: annotate code with **`yielding`** and **`atomic`** properties.

(Dynamic check: **`startAtomic()`**, **`endAtomic()`**, **`yield()`**)

Windows Fiber Programming

Example: Copy a File

```
typedef struct{
   DWORD dwFiberResultCode; // GetLastError() result code
   HANDLE hFile;            // handle to operate on
   DWORD dwBytesProcessed;  // number of bytes processed
} FIBERDATASTRUCT;
```

```
LPVOID g_lpFiber[FIBER_COUNT];
LPBYTE g_lpBuffer;
DWORD  g_dwBytesRead;
```

```
int __cdecl _tmain(int argc, TCHAR *argv[]){
   FIBERDATASTRUCT * fs = HeapAlloc(sizeof(FIBERDATASTRUCT) * FIBER_COUNT);
   // Allocate storage for the read/write buffer
   g_lpBuffer = (LPBYTE)HeapAlloc(GetProcessHeap(), 0, BUFFER_SIZE);
   fs[READ_FIBER].hFile = CreateFile(…); // Open source file
   fs[WRITE_FIBER].hFile = CreateFile(…); // Open destination file
   // Convert thread to a fiber, to allow scheduling other fibers
   g_lpFiber[PRIMARY_FIBER] = ConvertThreadToFiber(&fs[PRIMARY_FIBER]);
   // Create Read and Write fibers
   LPVOID read_fiber = CreateFiber(0,ReadFiberFunc,&fs[READ_FIBER]);
   LPVOID write_fiber = CreateFiber(0,WriteFiberFunc,&fs[WRITE_FIBER]);
   // Switch to the READ fiber
   SwitchToFiber(g_lpFiber[READ_FIBER]);
   // Here we have been scheduled again.
   printf("ReadFiber: result code is %lu, %lu bytes processed\n",
   fs[READ_FIBER].dwFiberResultCode, fs[READ_FIBER].dwBytesProcessed);
   printf("WriteFiber: result code is %lu, %lu bytes processed\n",
   fs[WRITE_FIBER].dwFiberResultCode, fs[WRITE_FIBER].dwBytesProcessed);
   <… clean up and return …>
}
```

Windows Fiber Programming

Example: Copy a File

```
VOID __stdcall ReadFiberFunc(LPVOID lpParameter){
   FIBERDATASTRUCT * fds = (FIBERDATASTRUCT*)lpParameter;
   fds->dwBytesProcessed = 0;

   while (1)   {
      // Read data from file specified in the READ_FIBER structure
      if (!ReadFile(fds->hFile, g_lpBuffer, BUFFER_SIZE, &g_dwBytesRead, NULL)){
         break;
      }
      // if we reached EOF, break
      if (g_dwBytesRead == 0) break;
      // Update number of bytes processed in the fiber data structure
      fds->dwBytesProcessed += g_dwBytesRead;
      // Switch to the write fiber
      SwitchToFiber(g_lpFiber[WRITE_FIBER]);
   } // while

   // Update the fiber result code
   fds->dwFiberResultCode = GetLastError();
   // Switch back to the primary fiber
   SwitchToFiber(g_lpFiber[PRIMARY_FIBER]);
}
```

Windows Fiber Programming

Example: Copy a File

```
VOID __stdcall WriteFiberFunc(LPVOID lpParameter){
   FIBERDATASTRUCT * fds = (FIBERDATASTRUCT*) lpParameter;
   DWORD dwBytesWritten;
   // Assume all writes succeeded.  If a write fails, the fiber
   // result code will be updated to reflect the reason for failure
   fds->dwBytesProcessed = 0;
   fds->dwFiberResultCode = ERROR_SUCCESS;
   while (1) {
      // Write data to the file specified in the WRITE_FIBER structure
      if (!WriteFile(fds->hFile, g_lpBuffer, g_dwBytesRead, &dwBytesWritten, NULL))
         break; // If an error occurred writing, break
      // Update number of bytes processed in the fiber data structure
      fds->dwBytesProcessed += dwBytesWritten;
      // Switch back to the read fiber
      SwitchToFiber(g_lpFiber[READ_FIBER]);
   } // while

   // If an error occurred, update the fiber result code...
   fds->dwFiberResultCode = GetLastError();
   // ...and switch to the primary fiber
   SwitchToFiber(g_lpFiber[PRIMARY_FIBER]);
}
```

Windows Fiber Programming

Example: Copy a File

```
voidDisplayFiberInfo(){
   FIBERDATASTRUCT * fds = (FIBERDATASTRUCT*) GetFiberData();
   LPVOID lpCurrentFiber = GetCurrentFiber();
   //
   // Determine which fiber is executing, based on the fiber address
   //
   if (lpCurrentFiber == g_lpFiber[READ_FIBER])
      printf("Read fiber entered");
   else   {
     if (lpCurrentFiber == g_lpFiber[WRITE_FIBER])
         printf("Write fiber entered");
     else
     {
         if (lpCurrentFiber == g_lpFiber[PRIMARY_FIBER])
            printf("Primary fiber entered");
         else
            printf("Unknown fiber entered");
  }
   }
   // Display dwParameter from the current fiber data structure
   printf(" (dwParameter is 0x%lx)\n", fds->dwParameter);
}
```

Integrating Thread and Fiber Programming

- **Question**: How to ensure that code written in one style can call code written in the other style?
- **Answer**: Adapters!
- **Example**:

manual

```
Certificate* GetCertData(User user) {
   // Look up certificate in the memory
   // cache and return the answer.
   // Else fetch from disk/network
   if (Lookup(user, cert))
      return certificate;
   certificate = DoIOAndGetCert();
   return certificate;
}
```

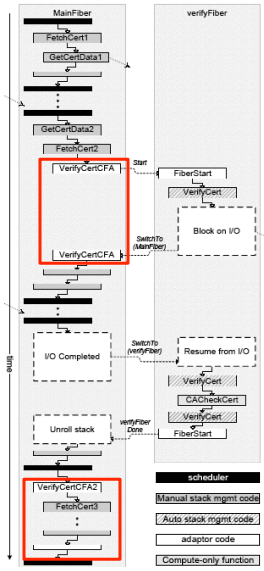
manual

```
bool FetchCert(User user, Certificate *cert) {
   // Get the certificate data from a
   // function that might do I/O
   certificate = GetCertData(user);
   if (!VerifyCert(user, cert)) {
      return false;
   }
}
```

automatic

```
bool VerifyCert(User user, Certificate * cert) {
   // Get the Certificate Authority (CA)
   // information and then verify certificate
   ca = GetCAInfo(cert);
   if (ca == NULL) return false;
   return CACheckCert(ca, user, cert);
}
```

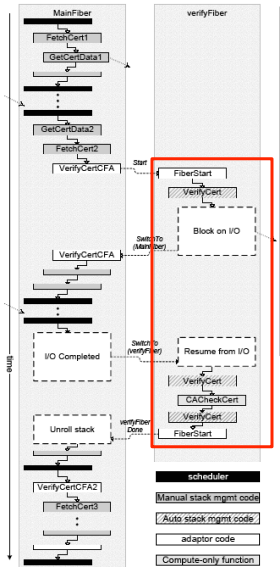
Fiber calls Thread




```
void VerifyCertCFA(CertData certData,
                   Continuation *callerCont) {
    // Executed on MainFiber
    Continuation *vcaCont =
            new Continuation(VerifyCertCFA2,
                             callerCont);
    Fiber *verifyFiber =
            new VerifyCertFiber(certData,
                                vcaCont);
    // On fiber verifyFiber, start executing
    // VerifyCertFiber::FiberStart
    SwitchToFiber(verifyFiber);
    // Control returns here when
    // verifyFiber blocks on I/O
}
```

```
void VerifyCertCFA2(Continuation *vcaCont) {
    // Executed on MainFiber.
    // Scheduled after verifyFiber is done
    Continuation *callerCont = vcaCont->arg1;
    callerCont->returnValue =vcaCont->returnValue;
    // "return" to original caller (FetchCert)
    (callerCont->function)(callerCont);
}
```

Fiber calls Thread (cont)




```
void VerifyCertCFA(CertData certData,
                   Continuation *callerCont) {
    // Executed on MainFiber
    Continuation *vcaCont =
            new Continuation(VerifyCertCFA2,
                             callerCont);
    Fiber *verifyFiber =
            new VerifyCertFiber(certData,
                                vcaCont);
    // On fiber verifyFiber, start executing
    // VerifyCertFiber::FiberStart
    SwitchToFiber(verifyFiber);
}
```

```
VerifyCertFiber::FiberStart() {
    // Executed on a fiber other than MainFiber
    // The following call could block on I/O.
    // Do the actual verification.
    this->vcaCont->returnValue =
                        VerifyCert(this->certData);
    // The verification is complete.
    // Schedule VerifyCertCFA2
    scheduler->schedule(this->vcaCont);
}
```