Asynchronous Input/Output

TOPICS

- Topic I Asynchronous I/O Overview
- Topic II Overlapped I/O
- Topic III Extended I/O

TOPIC I

Asynchronous I/O Overview

OVERVIEW

- Input and output are inherently slow compared to other processing due to delays:
  - Caused by track and sector seek time on random access devices
    - Discs and CD-ROMS
  - Caused by the relatively slow data transfer rate between a physical device and system memory
  - Waiting for data to be written into a pipe by another thread

Windows ASYNCHRONOUS I/O (1 of 3)

- Threads and processes
  - Each thread within a process (or in different processes) performs normal synchronous I/O
  - But other threads can continue execution
  - The parallel searching examples (`grepMP`, `grepMT`) used a form of thread asynchronous I/O

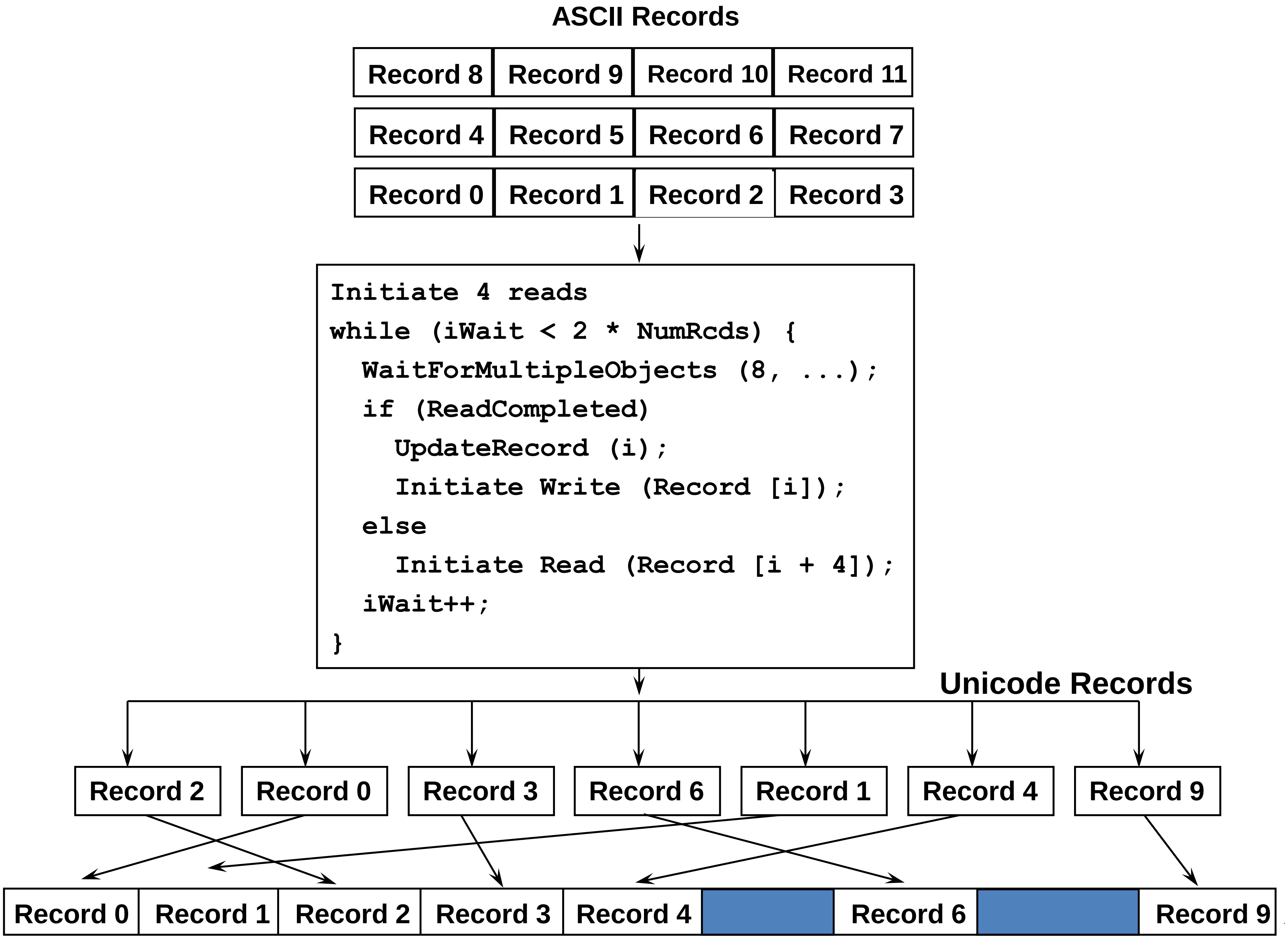
Windows ASYNCHRONOUS I/O (2 of 3)

- Overlapped I/O
  - A thread continues execution after issuing a read, write, or other I/O operation
  - The thread then waits on either the handle or a specified event when it requires the I/O results before continuing
  - Windows 9x supports overlapped I/O only for serial devices such as named pipes
    - Disc files are not supported

Windows ASYNCHRONOUS I/O (3 of 3)

- Completion routines
  - Also called extended I/O or alertable I/O
  - The system invokes a specified “completion routine” within the thread when the I/O operation completes
  - Windows 9x only supports serial devices

AN ASYNCHRONOUS FILE UPDATE MODEL



ASCII Records

Record 8 | Record 9 | Record 10 | Record 11

Record 4 | Record 5 | Record 6 | Record 7

Record 0 | Record 1 | Record 2 | Record 3

```
Initiate 4 reads
while (iWait < 2 * NumRcds) {
  WaitForMultipleObjects (8, ...);
  if (ReadCompleted)
    UpdateRecord (i);
    Initiate Write (Record [i]);
  else
    Initiate Read (Record [i + 4]);
  iWait++;
}
```

Unicode Records

Record 2 | Record 0 | Record 3 | Record 6 | Record 1 | Record 4 | Record 9

Record 0 | Record 1 | Record 2 | Record 3 | Record 4 | | Record 6 | | Record 9

TOPIC II

Overlapped I/O

OVERLAPPED I/O (1 of 2)

- Overlapped structures are options on four I/O functions that can potentially block while the operation completes:
  - `ReadFile`
  - `WriteFile`
  - `TransactNamedPipe`
  - `ConnectNamedPipe`

OVERLAPPED I/O (2 of 2)

- You must specify `FILE_FLAG_OVERLAPPED` as part of `fdwAttrsAndFlags` (for `CreateFile`)
- Or as part of `fdwOpenMode` (for `CreateNamedPipe`)
  - Doing so specifies that the pipe or file is to be used only in overlapped mode
- Note: Overlapped I/O can be used with Windows Sockets

OVERLAPPED I/O CONSEQUENCES

- I/O operations do not block
- The system returns immediately from these calls:

```
ReadFile
WriteFile
TransactNamedPipe
ConnectNamedPipe
```

- The returned function value is not useful to indicate success or failure
- The I/O operation is most likely not yet complete

CONSEQUENCES (2 of 2)

- The returned number of bytes transferred is also not useful
- The program may issue multiple reads or writes on a single file handle
  - The file pointer is meaningless
- The program must be able to wait on (synchronize with) I/O completion
  - In case of multiple outstanding operations on a single handle, it must be able to determine which operation completed
  - I/O operations do not necessarily complete in the same order as they were issued

OVERLAPPED STRUCTURES (1 of 4)

- The overlapped structure:
  - Indicates the file position (64 bits)
  - Indicates the event that will be signalled when the operation completes
  - Specified by the `lpOverlapped` parameter of `ReadFile` (for example)
- There are also two internal `DWORD`s that the programmer does not use

OVERLAPPED STRUCTURES (2 of 4)

```
typedef struct _OVERLAPPED {
    DWORD Internal;
    DWORD InternalHigh;
    DWORD Offset;
    DWORD OffsetHigh;
    HANDLE hEvent;
} OVERLAPPED
```

OVERLAPPED STRUCTURES (3 of 4)

- The file position (pointer) must be set in `Offset` and `OffsetHigh`
- `hEvent` is an event handle
  - Created with `CreateEvent`
  - The event can be named or unnamed, but it must be a manually reset event
- `hEvent` can be `NULL`, in which case you can wait on the file handle (also a synchronization event)
- The system signals completion on the file handle when `hEvent` is `NULL`

OVERLAPPED STRUCTURES (4 of 4)

- This event is immediately reset by the system when the program makes an I/O call
    - Set to the non-signaled state
- When the I/O operation completes, the event is signaled
- You can still use the overlapped structure as an alternative to `SetFilePointer`
    - Even if the file handle is synchronous
- An outstanding I/O operation is uniquely identified by the combination of file handle and overlapped structure

CAUTIONS

- Do not reuse an `OVERLAPPED` structure while its associated I/O operation, if any, is outstanding
- Do not reuse an event while it is part of an overlapped structure
- If you have more than one outstanding request on an overlapped handle, use events for synchronization rather than the file handle
- If the overlapped structure or event are automatic variables in a block, be certain that you do not exit the block before synchronizing with the I/O operation

OVERLAPPED I/O STATES (1 of 3)

- An overlapped `ReadFile` or `WriteFile` operation returns immediately
  - In most cases, the I/O will not be complete
  - The read or write returns a `FALSE`
  - `GetLastError ()` will return `ERROR_IO_PENDING`
- `GetOverlappedResult` allows you to determine how many bytes were transferred

OVERLAPPED I/O STATES (2 of 3)

```
BOOL GetOverlappedResult (HANDLE hFile,
    LPOVERLAPPED lpoOverlapped,
    LPWORD lpcbTransfer,
    BOOL fWait)
```

OVERLAPPED I/O STATES (3 of 3)

- fWait, if TRUE, specifies that GetOverlappedResult will wait until specified operation completes
  - Otherwise, return immediately
- The function returns TRUE only if the operation has completed
- GetLastError will return ERROR_IO_INCOMPLETE in case of a FALSE return from GetOverlappedResult
- The actual number of bytes transferred is in *lpcbTransfer

TOPIC III

Extended I/O

EXTENDED I/O WITH COMPLETION

- Rather than requiring a thread to wait for a completion signal on an event or handle, the system can invoke a user-specified “completion routine” when an I/O operation completes
  - Use a family of “extended” I/O functions identifiable with the `Ex` suffix:

    ```
    ReadFileEx
    WriteFileEx
    ```

EXT. I/O WITH COMPLETION (2 of 2)

- Additionally, use one of the three “alertable wait” functions:

  ```
  WaitForSingleObjectEx
  WaitForMultipleObjectsEx
  SleepEx
  ```

- Extended I/O is sometimes called “alertable I/O”

EXTENDED READ

```
• BOOL ReadFileEx (HANDLE hFile,
•     LPVOID lpBuffer,
•     DWORD nNumberOfBytesToRead,
•     LPOVERLAPPED lpOverlapped,
•     LPOVERLAPPED_COMPLETION_ROUTINE lpcr)
```

EXTENDED WRITE

- BOOL WriteFileEx (HANDLE hFile,
- LPVOID lpBuffer,
- DWORD nNumberOfBytesToWrite,
- LPOVERLAPPED lpOverlapped,
- LPOVERLAPPED_COMPLETION_ROUTINE lpcr)

EXTENDED READ AND WRITE (1 of 4)

- The extended read and write functions can be used with open file and named pipe (or even mailslot) handles if `FILE_FLAG_OVERLAPPED` was used at open (or create) time
- The two functions are familiar but have an extra parameter to specify the completion routine
- The overlapped structures must be supplied
  - There is no need to specify the `hEvent` member

EXTENDED READ AND WRITE (2 of 4)

- The extended functions do not require the parameters for the number of bytes transferred
- That information is conveyed to the completion routine, which must be included in the program
- The completion routine has parameters for the byte count, an error code, and the overlapped structure

EXTENDED READ AND WRITE (3 of 4)

- VOID WINAPI *FileIOCompletionRoutine* (
- DWORD fdwError, DWORD cbTransferred,
- LPOVERLAPPED lpo)
  - FileIOCompletionRoutine is a place holder, not an actual function name
  - fdwError is limited to 0 (success) and ERROR_HANDLE_EOF (when a read tries to read past the end-of-file)

EXTENDED READ AND WRITE (4 of 4)

- Two things must happen before the completion routine is invoked by the system:
  - The actual I/O operation must complete
  - The calling thread must be in an “alertable wait” state
- To get into an alertable wait state, a thread must make an explicit call to one of three alertable wait functions

ALERTABLE WAIT FUNCTIONS

- `DWORD WaitForSingleObjectEx (`
- `    HANDLE hObject, DWORD dwTimeOut,`
- `    BOOL fAlertable)`

- `DWORD WaitForMultipleObjectsEx (`
- `    DWORD cObjects, LPHANDLE lphObjects,`
- `    BOOL fWaitAll, DWORD dwTimeOut`
- `    BOOL fAlertable)`

- `DWORD SleepEx (DWORD dwTimeOut,`
- `    BOOL fAlertable)`

ALERTABLE WAIT (2 OF 2)

- Each has an `fAlertable` flag which must be set to `TRUE`
- These three functions will return as soon as one of the following occurs:
  - Handle(s) are signaled so as to satisfy one of the first two functions
  - Any completion routine in the thread finishes
  - The time period expires (it could be `infinite`)
- There are no events associated with the `ReadFileEx` and `WriteFileEx` overlapped structures
- `SleepEx` is not associated with a synchronization object

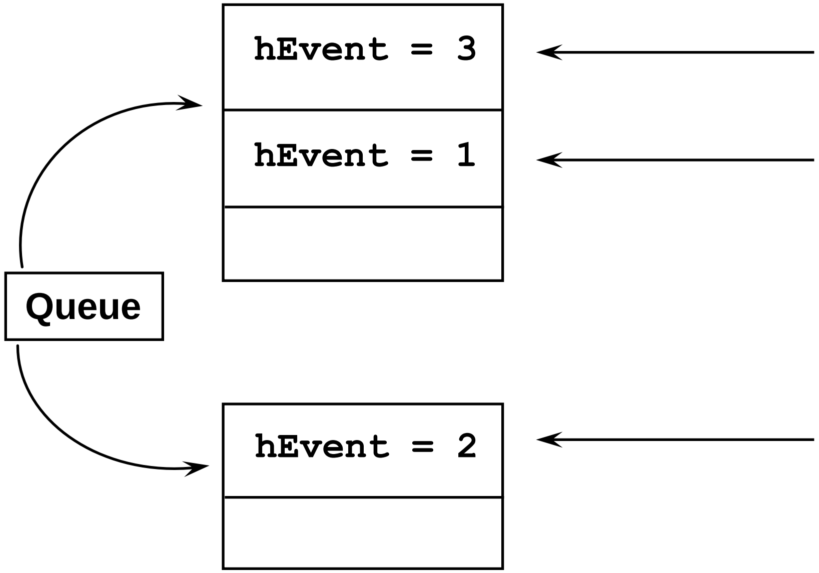
EXECUTION OF COMPLETION ROUTINES (1 of 2)

- As soon as an extended I/O operation is complete, its associated completion routine is queued for execution
- All of a thread’s queued completion routines will be executed when the thread enters an alertable wait state, and the alertable wait function returns only after the completion routines return

EXECUTION OF COMPLETION ROUTINES (2 of 2)

- `SleepEx` will return `WAIT_IO_COMPLETION` if one or more queued completion routines were executed
- `GetLastError` will return this same value after one of the wait functions returns
- You can use a `0` timeout value with any alertable wait function, and it will return immediately
- You can use the `hEvent` data member of the overlapped structure to convey any information you wish to the completion routine

ASYNCHRONOUS I/O WITH COMPLETION ROUTINES



```
hIn = CreateFile (... FILE_FLAG_OVERLAPPED ...);
for (i = 0; i < 3; i++) {
    ov [i].hEvent = i + 1;
    ov [i].Offset = i * LSIZE;
    ReadFileEx (hIn, &ov [i], RDone);
}
        /* More thread code */
    [Third read completes]
        /* More thread code */
    [First read completes]
        /* More thread code */
    SleepEx (INFINITE);
    [Completion Routine (RDone) executes twice]
    [Return from SleepEx]
        /* More thread code */
    [Second read completes]
        /* More thread code */
    SleepEx (INFINITE);
    [Completion Routine (RDONE) executes once]
        /* More thread code */
    ExitProcess (0);
RDone (... lpov ...)
{   /* Indicate I/O complete */
        CompleteFlag [lpov -> hEvent] = TRUE;
}
```

SUMMARY (1 of 2)

- Windows gives you three methods for performing asynch-ronous I/O. Each technique has its own advantages and unique characteristics. The choice is often a matter of individual preference.
  - Using threads is the most general technique
    - Works with Windows 9x
    - Each thread is responsible for a sequence of one or more synchronous, blocking I/O operations
    - Each thread should have its own file or pipe handle

SUMMARY (2 of 2)

- Overlapped I/O allows a single thread to perform asynchronous operations on a single file handle
  - You need an event handle for each operation, rather than a thread and file handle pair
  - You must wait specifically for each I/O operation to complete and then perform any required clean-up or bookkeeping operations
- Extended I/O automatically invokes the completion code
  - It does not require additional events