Asynchronous I/O

Asynchronous I/O

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Synchronous I/O

- All previously analyzed I/O operations are thread-synchronous
  - I/O is **blocking** and the thread waits until the I/O operation completes
- Unfortunately, I/O operations are inherently **slow** compared to other processing
  - Delays may be caused by
    - Hardware device, e.g., track and sector seek time on random access, etc.
    - Relatively slow data transfer rate between a physical device and the system memory
    - Network transfer using file servers, storage area networks, etc.

Asynchronous I/O

- Threads can perform asynchronous I/O
  - A thread can continue **without** waiting for an I/O operation to complete
- Windows' OS has three methods for performing asynchronous I/O
  - Each technique has its own advantages and unique characteristics
  - The choice is often a matter of individual preference

Asynchronous I/O

1 ➢ Multithread I/O

- Each thread within a process (or in different processes) may perform normal synchronous I/O
  - 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
- **Other** threads can **continue** execution
- This is the most general technique

Asynchronous I/O

The one we focus on

2 ➢ Overlapped I/O with waiting

- A thread **continues** execution after issuing a read, write, or another I/O operation
- When the thread requires the I/O results before continuing it **awaits** on either the I/O handle or a specified event

3 ➢ Extended (or alertable) I/O with completion routines

- The system invokes a specified "completion routine" **callback funtion** within the thread when the I/O operation completes
- Extended I/O require extended I/O functions (such as **ReadFileEx** and **WriteFileEx**)

Overlapped I/O

- Overlapped I/O with waiting uses the **overlapped** data structure to implement aysncronous functions
  - First, specify the FILE_FLAG_OVERLAPPED flag as part of fdwAttrsAndFlags for CreateFile
    - It specifies that the file is to be used only in overlapped mode
  - Then, use the overlapped data structure with
    - ReadFile and WriteFile
    - Use the **file handle** or the **overlapped event** to wait for

Use the handle for **single**, the event for **multiple** I/O calls

```
type def struct _OVERLAPPED {
  DWORD Internal;
  DWORD InternalHigh;
  DWORD Offset;
  DWORD OffsetHigh;
  HANDLE hEvent;
] OVERLAPPED;
```

Used for asynchronous I/O

I/O Functions

- ReadFile and WriteFile can potentially block while the operation completes but with the overlapped data structure they are asynchronous
  - I/O operations do not block
    - The system returns immediately from a call to ReadFile and WriteFile
  - The **returned** function **value** is not useful to indicate success or failure
    - A FALSE value in return does not necessarily indicate a failure, because
    - The I/O operation is most likely not yet complete
      - In this case GetLastError will return the value ERROR_IO_PENDING

I/O Functions

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

The handle is the same

The ov data structure differs

GetOverlappedResult

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

Be certain lpOverlapped is unchanged from when it was used with the overlapped I/O operation

- **After** waiting on a synchronization object
  - GetOverlappedResult allows you to determine how many bytes were transferred
- Parameter
  - The handle and the lpOverlapped structure combine to indicate the **specific** I/O operation

GetOverlappedResult

- **lpcbTransfer**
  - The actual number of bytes transferred
- **fWait**
  - If TRUE, it specifies that GetOverlappedResult will wait until the specified operation completes
  - Otherwise, it will return immediately

- Return value
  - TRUE, only if the operation has completed

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

Example

Synchronization on a file handle (**single** I/O op)

Overlapped I/O is simple when there is only one outstanding operation

```
OVERLAPPED ov = { 0, 0, 0, 0, NULL };
HANDLE hF;
DWORD nREAD;
record_t r;
. . .
hF = CreateFile (..., FILE_FLAG_OVERLAPPED, ...);
ReadFile (hF, &r, sizeof(record_t), &nR, &ov);
...
Perform other processing
nR is probably not valid
...
WaitForSingleObject (hF, INFINITE);
GetOverlappedResult (hF, &ov, &nR, FALSE);
```

Wait-for the operation to end on the file handle

Get ReadFile result

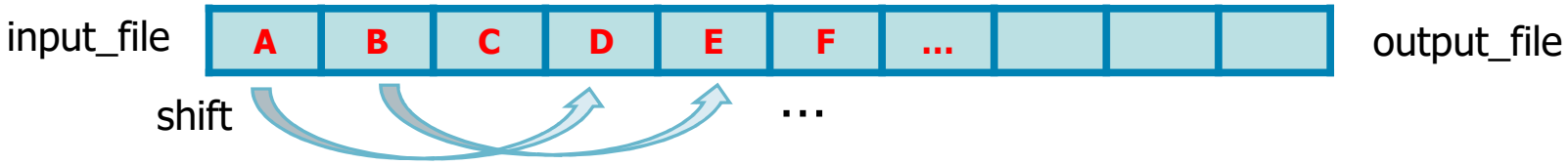
#Bytes read

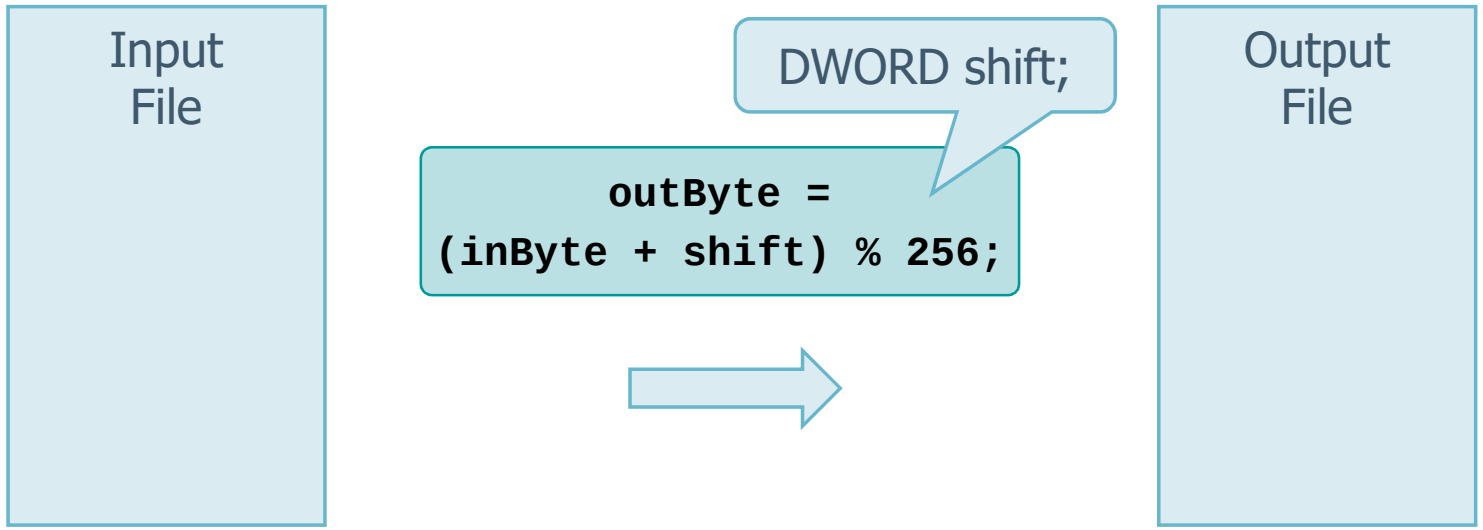
Example

Synchronization on the event (**multiple** I/O op)

Overlapped I/O is more complex when there is more than one outstanding operation

- Perform the following file encryption
  - Caesar's cipher (circa 50 BC)
    - pgrm_name shift input_file output_file





Solution 1 & 2

- Solution 1
  - Sequential, byte by byte
- Solution 2
  - Sequential, record by record

Selected experimentally to optimize performace

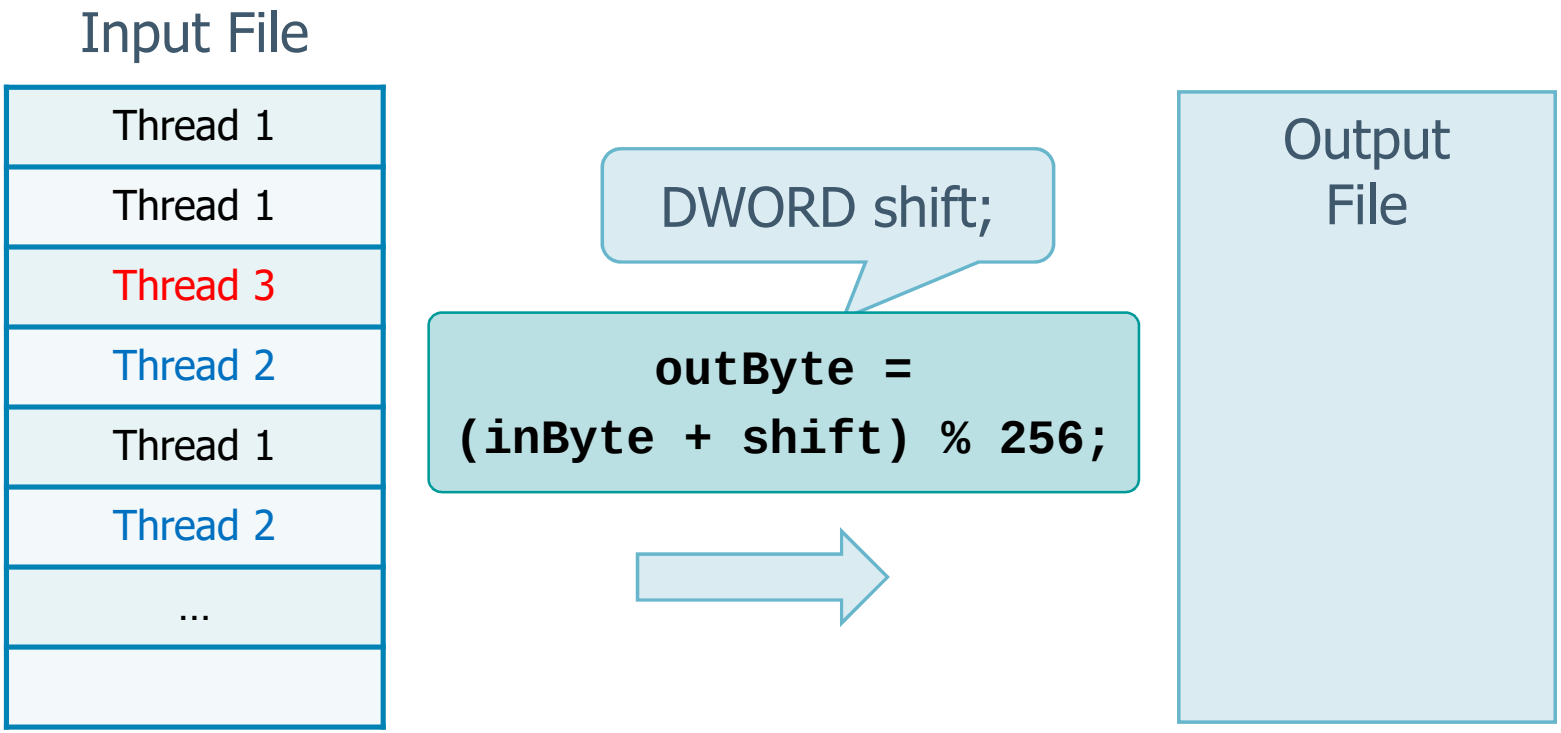
```
while (ReadFile (hIn, buffer, BUF_SIZE, &nIn, NULL)
  && nIn > 0 && WriteOK) {

  for (iCopy=0; iCopy<nIn; iCopy++) {
    buffer[iCopy] = (buffer[iCopy] + shift) % 256;
  }

  writeOK = WriteFile (hOut, buffer, nIn, &nOut, NULL);
}
```

Encryption

Solution 3A

- Solution 3A
  - Parallel, with N threads
    - Let the threads run freely (**dynamic** partition)
      - More contention

The faster thread gets next record



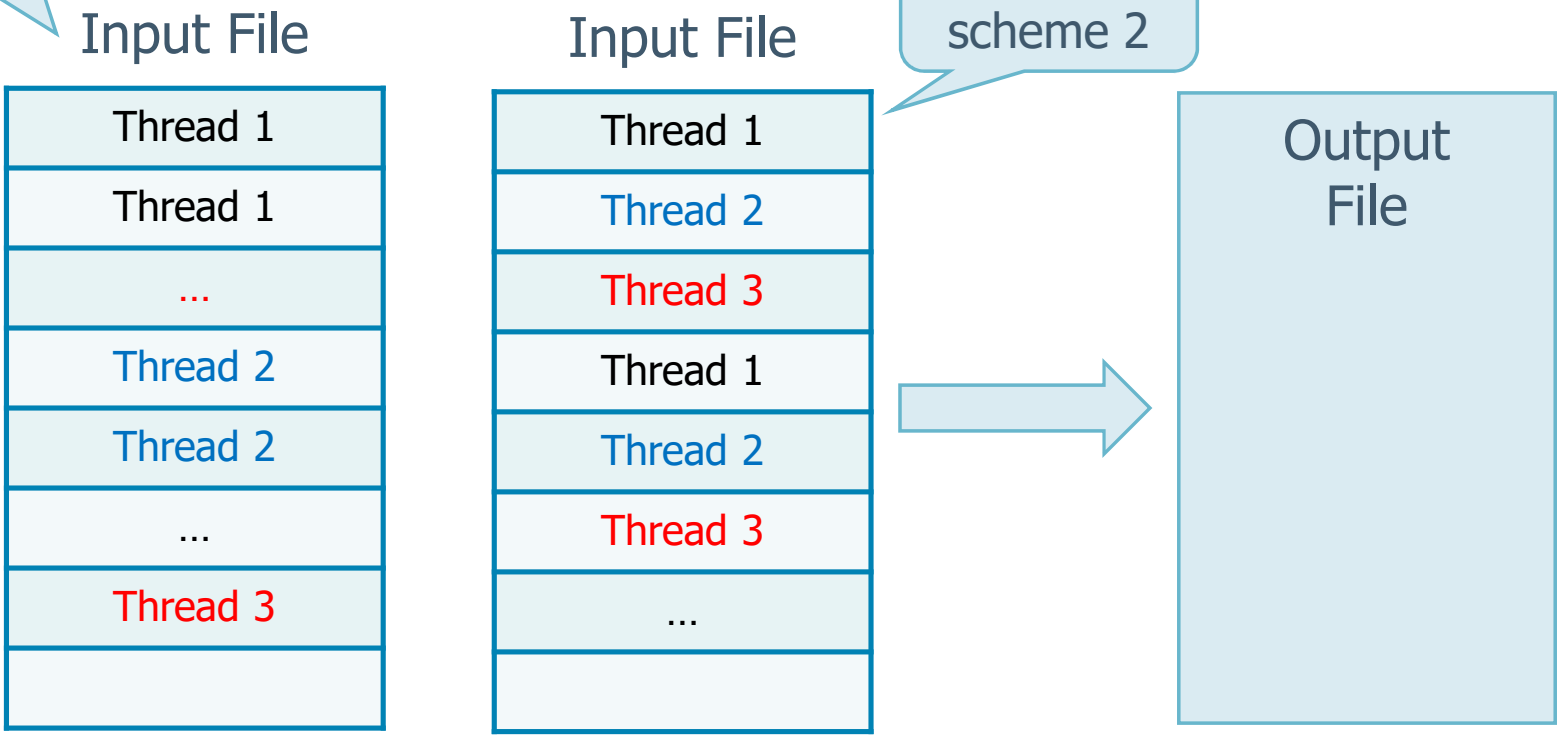
Solution 3B

- Solution 3B
  - Parallel, with N threads
    - Assign to each thread 1/N of the file (**static** partition)
      - Efficiency is limited by the slower thread

Each one gets its own part of the file

Partition scheme 1

Partition scheme 2



Solution 4

Solution 4

- Use memory mapped files

Header inclusion

Input file

Output file

Encryption constant

```
#include ...

VOID caesarCipher (LPCTSTR fIn, LPCTSTR fOut, DWORD shift) {
  BOOL complete = FALSE;
  HANDLE hIn = INVALID_HANDLE_VALUE;
  HANDLE hOut = INVALID_HANDLE_VALUE;
  HANDLE hInMap = NULL, hOutMap = NULL;
  LPTSTR pIn = NULL, pInFile = NULL;
  LPTSTR pOut = NULL, pOutFile = NULL;
  LARGE_INTEGER fileSize;
```

Variable definitions

Solution 4

To avoid problems with large file it is possible to map one block at a time

```
hIn = CreateFile (fIn, GENERIC_READ, 0, NULL,
   OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
GetFileSizeEx (hIn, &fileSize);
if (fileSize.HighPart > 0)
  ... This file is too large to map on a Win32 system ...
hInMap = CreateFileMapping (hIn, NULL, PAGE_READONLY,
  0, 0, NULL);
pInFile = MapViewOfFile (hInMap, FILE_MAP_READ, 0, 0, 0);

hOut = CreateFile (fOut, GENERIC_READ | GENERIC_WRITE,
  0, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
hOutMap = CreateFileMapping (hOut, NULL, PAGE_READWRITE,
  fileSize.HighPart, fileSize.LowPart, NULL);
pOutFile = MapViewOfFile (hOutMap, FILE_MAP_WRITE, 0, 0,
  (SIZE_T)fileSize.QuadPart);
```

Open and map entire input file

Open and map entire output file

Solution 4

```
  pIn = pInFile;
  pOut = pOutFile;
  while (pIn < pInFile + fileSize.QuadPart) {
    *pOut = (*pIn + shift) % 256;
    pIn++; pOut++;
  }
  UnmapViewOfFile (pOutFile);
  UnmapViewOfFile (pInFile);
  CloseHandle (hOutMap);
  CloseHandle (hInMap);
  CloseHandle (hIn);
  CloseHandle (hOut);

  return;
}
```

Encrypt file

Clean and close

Solution 5

- Solution 5
  - Use an asynchronous file update model

```
Initiate 4 reads
while (all records have been encoded) {
  WaitForMultipleObjects (8, ...);
  if (ReadCompleted)
    UpdateRecord (i);
    Initiate Write (Record [i]);
  else
    Initiate Read (Record [i + 4]);
  n_record++;
}
```

Perform 4 Read in "parallel"

Wait for 1 out of 8 events 4 ReadFile + 4 WriteFile

Encryption

Next write

Next read

Solution 5

#Read in "parallel"

Selected experimentally to optimize performace

```
#include ...

#define MAX_OVRLP    4
#define REC_SIZE  8192

int _tmain (int argc, LPTSTR argv[]) {
  HANDLE hInputFile, hOutputFile;
  DWORD shift, nIn[MAX_OVRLP], nOut[MAX_OVRLP], ic, i;
  OVERLAPPED overLapIn[MAX_OVRLP], overLapOut[MAX_OVRLP];
  HANDLE hEvents[2][MAX_OVRLP];
  CHAR buffer[MAX_OVRLP][REC_SIZE], cShift;
  LARGE_INTEGER curPosIn, curPosOut, fileSize;
  LONGLONG nRecords, iWaits;

  shift = _ttoi(argv[1]);
```

Cipher shift

Solution 5

```
hInputFile = CreateFile (argv[2], GENERIC_READ,
  0, NULL, OPEN_EXISTING, FILE_FLAG_OVERLAPPED, NULL);
hOutputFile = CreateFile (argv[3], GENERIC_WRITE,
  0, NULL, CREATE_ALWAYS, FILE_FLAG_OVERLAPPED, NULL);
if (hInputFile==INVALID_HANDLE_VALUE ||
 hOutputFile==INVALID_HANDLE_VALUE) {
  ...
}

GetFileSizeEx (hInputFile, &fileSize);
nRecords = (fileSize.QuadPart + REC_SIZE - 1) / REC_SIZE;
```

Open I and O files

Compute number of records (including reamining bytes)

Solution 5

```
curPosIn.QuadPart = 0;
for (ic=0; ic<MAX_OVRLP; ic++) {
  hEvents[0][ic] = overLapIn[ic].hEvent =
    CreateEvent (NULL, TRUE, FALSE, NULL);
  hEvents[1][ic] = overLapOut[ic].hEvent =
    CreateEvent (NULL, TRUE, FALSE, NULL);
  overLapIn[ic].Offset = curPosIn.LowPart;
  overLapIn[ic].OffsetHigh = curPosIn.HighPart;
  if (curPosIn.QuadPart < fileSize.QuadPart) {
    ReadFile (hInputFile, buffer[ic], REC_SIZE,
      &nIn[ic], &overLapIn[ic]);
  }
  curPosIn.QuadPart += (LONGLONG)REC_SIZE;
}
```

For each buffer

Create manual-reset unsignaled events

Assign these events to the ov data structure

Initiate a read on each buffer

Solution 5

```
iWaits = 0;
while (iWaits < 2 * nRecords) {
  ic = WaitForMultipleObjects (2 * MAX_OVRLP,
    hEvents[0], FALSE, INFINITE) - WAIT_OBJECT_0;
  iWaits++;
  if (ic < MAX_OVRLP) {
    GetOverlappedResult (hInputFile, &overLapIn[ic],
      &nIn[ic], FALSE);
    ResetEvent (hEvents[0][ic]);
    curPosIn.LowPart = overLapIn[ic].Offset;
    curPosIn.HighPart = overLapIn[ic].OffsetHigh;
    curPosOut.QuadPart = curPosIn.QuadPart;
    overLapOut[ic].Offset = curPosOut.LowPart;
    overLapOut[ic].OffsetHigh = curPosOut.HighPart;
```

While read and write operations are running (until the end of file)

Wait for a read or a write to complete

If a read completed

Record Position (in)

Reset event before next WFMO

Record Position (out)

Set record Position into ov

Process record and start a write in the same position

Solution 5

```
    for (i=0; i<nIn[ic]; i++)
      buffer[ic][i] = (buffer[ic][i] + Shift) % 256;
    WriteFile (hOutputFile, buffer[ic], nIn[ic],
      &nOut[ic], &overLapOut[ic])
    curPosIn.QuadPart += REC_SIZE * (LONGLONG) (MAX_OVRLP);
    overLapIn[ic].Offset = curPosIn.LowPart;
    overLapIn[ic].OffsetHigh = curPosIn.HighPart;
  } else
    if (ic < 2 * MAX_OVRLP) {
      ic -= MAX_OVRLP;
      GetOverlappedResult (hOutputFile, &overLapOut[ic],
        &nOut[ic], FALSE)) {
      ResetEvent (hEvents[1][ic]);
      curPosIn.LowPart = overLapIn[ic].Offset;
      curPosIn.HighPart = overLapIn[ic].OffsetHigh;
```

Encrypt the record

Write it

Prepare overlapped for next read

If a write completed

Start a new read

Solution 5

```
      if (curPosIn.QuadPart < fileSize.QuadPart) {
        ReadFile (hInputFile, buffer[ic], REC_SIZE,
          &nIn[ic], &overLapIn[ic]);
      }
    } else { ... Error ... }
  }
  for (ic = 0; ic < MAX_OVRLP; ic++) {
    CloseHandle (hEvents[0][ic]);
    CloseHandle (hEvents[1][ic]);
  }
  CloseHandle (hInputFile);
  CloseHandle (hOutputFile);
  return 0;
}
```

No read and no write WFMO error

Close handles and quit

Performace

From J. Hart (2010)
Chapter 14

640MB file

Bare encryption

Memory mapped files

Asynchronous I/O

File Checking

Decryption