**MP3:** Kaynak sesi kayıplı sıkıştırma algoritmasıyla kodlayan ve kendi format bilgilerini (bitrate, sample rate vb.) içeren bir **header** ile saklayan ses formatıdır.  
**BIN:** Ham (işlenmemiş) veri dosyasıdır, genellikle **header içermez** ve format tanımı olmadan sadece bitlerden oluşur.  
**WAV:** Genelde sıkıştırılmamış PCM verisini saklar ve ses parametrelerini (sample rate, kanal sayısı, bit derinliği vb.) tanımlayan **44 byte’lık bir WAV header** içerir.

**Fark:** MP3 ve WAV dosyalarında **header** bulunarak veri formatı tanımlanır, MP3 kayıplı ve sıkıştırılmıştır, WAV ham ve yüksek kalitelidir, BIN ise genelde header içermeyen, ham ve tanımsız veridir.

1. **Memory-Mapped I/O (CreateFileMapping + MapViewOfFile):** Dosya içeriğini RAM’e doğrudan haritalayıp read() yerine bellek pointer’ı üzerinden erişim sağlar.
2. **WAV Header Yazımı:** WAV dosyası formatını tanımlayan 44 byte’lık standart metadata kısmını oluşturur.
3. **FILE\_FLAG\_NO\_BUFFERING:** Windows dosya önbelleğini bypass ederek veriyi doğrudan diske yazar, ek kopyalama maliyetini azaltır.
4. **FILE\_FLAG\_SEQUENTIAL\_SCAN:** İşletim sistemine dosyanın sıralı okunup yazılacağını bildirerek disk I/O optimizasyonu yapar.
5. **Double-Buffering:** İki buffer kullanarak biri diske yazılırken diğerinin doldurulmasını sağlar ve CPU ile I/O’yu pipeline eder.
6. **QueryPerformanceCounter:** Yüksek çözünürlüklü zaman ölçümü yaparak işlem süresini doğru şekilde hesaplar.

**Normal (Basit) Kodda Ne Olur?**

Normal bir dosya dönüştürme kodu şöyle çalışır:

1. std::ifstream ile küçük bir buffer (genelde 4KB-8KB) kullanarak dosyadan veri okunur.
2. std::ofstream ile yine 4KB-8KB bloklar halinde dosyaya yazılır.
3. Her okuma ve yazma işlemi **read() / write() sistem çağrıları** ile kernel'e gider.

**Bilgisayarın Kullandığı Kaynaklar:**

* **CPU:** Sık sık read() ve write() çağrısı yapmak CPU'yu meşgul eder. (Syscall overhead)
* **RAM:** Küçük buffer, veriyi kernel cache (disk önbelleği) üzerinden alır.
* **Disk:** Çok fazla küçük I/O operasyonu olduğundan disk throughput verimli kullanılmaz.

**Sorun:** Bu yöntem büyük dosyalarda çok yavaştır çünkü CPU, disk I/O için bekler.

**Bizim Kullandığımız Teknolojiler ve Mantıkları**

**1. Memory-Mapped I/O (CreateFileMapping + MapViewOfFile)**

* Normalde dosya okumak için read() çağrısı yaparsın. Bu, kernel’den user-space'e veri kopyalar.
* **Memory-mapped I/O** (mmap benzeri) ile **dosya içeriği RAM adres alanına direkt "map" edilir.**
* Artık pcm\_data[i] gibi pointer üzerinden **diziden veri okur gibi** dosya içeriğine erişiyoruz.

**Bilgisayarın Kullandığı Kaynaklar:**

* **CPU:** read() syscall çağrısı yok, memcpy ile direkt bellekte çalışıyor. Syscall overhead sıfıra yakın.
* **RAM:** Kernel ve user-space arasında ekstra kopyalama yok, tek kopya.
* **Disk:** Disk, gerekli sayfaları otomatik olarak (page fault ile) RAM'e yükler.

**Kıyas:**  
Normal kodda 4KB-8KB okuma yapılırken, burada diskten direkt **sayfa bazlı (genelde 4MB chunk)** okuma yapılıyor.

**2. Büyük Buffer (128 MB)**

* Normal kodda std::ofstream default olarak 4KB buffer kullanır.
* Biz **128 MB** gibi dev bir buffer kullanıyoruz.

**Bilgisayarın Kullandığı Kaynaklar:**

* **CPU:** WriteFile() çağrısı daha az (örneğin 1.5GB dosyada 12 çağrı), overhead düşüyor.
* **Disk:** Disk, büyük ve sıralı veri yazımında **maksimum hızına yakın** çalışıyor.
* **RAM:** 128MB RAM buffer ayrılıyor, ama bu kabul edilebilir.

**Kıyas:**  
Normal kodda binlerce küçük write olur, bizde sadece birkaç dev write çağrısı var.

**3. FILE\_FLAG\_NO\_BUFFERING**

* Windows’un disk cache’ini atlıyoruz. Yani veri **direkt diske** yazılıyor.
* Disk cache genelde faydalıdır, ama **büyük dosyalarda yavaşlar** (çünkü cache flush yapmak zaman alır).

**Bilgisayarın Kullandığı Kaynaklar:**

* **CPU:** Kernel tarafında disk cache yönetimi yok, daha az yük.
* **Disk:** Veri doğrudan diske gidiyor, yazma süreci net olarak disk hızına bağlı.
* **RAM:** Kernel cache kullanmadığı için RAM'de ekstra yer kaplamıyor.

**Kıyas:**  
Normal kodda OS cache’ye yazıp sonra diske flush eder. Biz doğrudan diske yazıyoruz. Bu sayede 2 kez kopyalama yapmıyoruz ve cache flushı(cachenin boşaltılmasını) beklemek zorunda kalmıyoruz.

**4. FILE\_FLAG\_SEQUENTIAL\_SCAN**

* Windows’a dosyanın baştan sona sıralı yazılacağını söylüyoruz.
* Disk scheduler, okuma/yazma optimizasyonunu bu bilgiye göre yapıyor.

**Bilgisayarın Kullandığı Kaynaklar:**

* Disk ve kernel bu ipucuyla **okuma/yazma sıradüzenini optimize ediyor.**

**Kıyas:**  
Normal kodda OS bu bilgiyi bilmediği için rastgele erişim gibi optimize etmeye çalışır.

**5. Double-Buffering**

* İki ayrı buffer kullanıyoruz: Buffer A ve Buffer B.
* **Mantık:**
  + Buffer A diske yazılırken, CPU aynı anda Buffer B’yi memcpy ile dolduruyor.
  + Disk ve CPU beklemeden **pipeline** mantığında paralel çalışıyor.

**Bilgisayarın Kullandığı Kaynaklar:**

* **CPU:** memcpy yaparken disk bir önceki veriyi yazıyor.
* **Disk:** CPU'dan boşuna veri beklemiyor.
* **RAM:** 2 × 128 MB = 256 MB RAM kullanılır, ama verim çok artar.

**Kıyas:**  
Normal kodda tek buffer olduğundan, **disk veya CPU boşta bekleyebilir.**

