Lecture 7: Inter-Process Communication
Shared Memory

Learning Goals

- Explain *why* processes cannot share local memory as easily as threads
- Describe a possible mechanism of how processes *can* share blocks of memory
- Able to create and apply *named* **shared memory objects** for inter-process communication

Interprocess Communication (IPC)

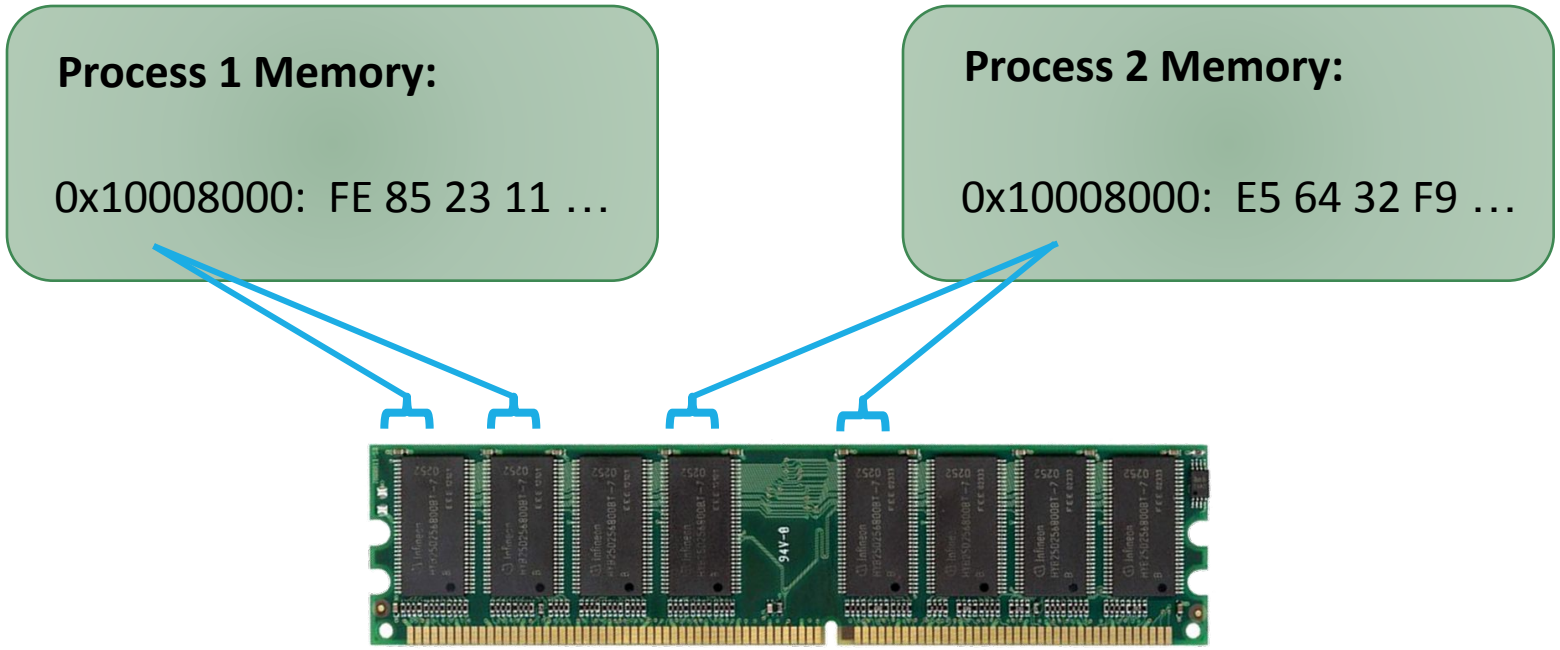
For processes to work together in a **cohesive** way to accomplish a **unified goal** (i.e. System Software), they need to be able to **communicate** with each other.

Thread communication is pretty straight-forward, since threads share memory. Processes ***do not***. Thus, we need some other mechanisms for sharing information.

Communication between processes on *different* machines present even further challenges, since now they need to communicate without sharing any hardware resources, or even operating systems. We can no longer rely on operating system kernel calls for this.

Memory Access

Unlike threads within a single process, separate processes **do not** share **address space**.



Memory Protection

Furthermore, most operating systems make use of a hardware-based **Memory Management Unit (MMU)** to enforce protection of process memory.

This chip prevents your program accessing random memory that has not been explicitly allocated to it by the OS, generating an Exception or Access violation if your process tries to reach outside of its bounds.

The only case where memory *can* be directly shared between processes is if you're using a special-purpose device or chip (e.g. in embedded systems). In such cases, that device's memory space is no longer controlled by the OS at all (that now becomes *your* job).

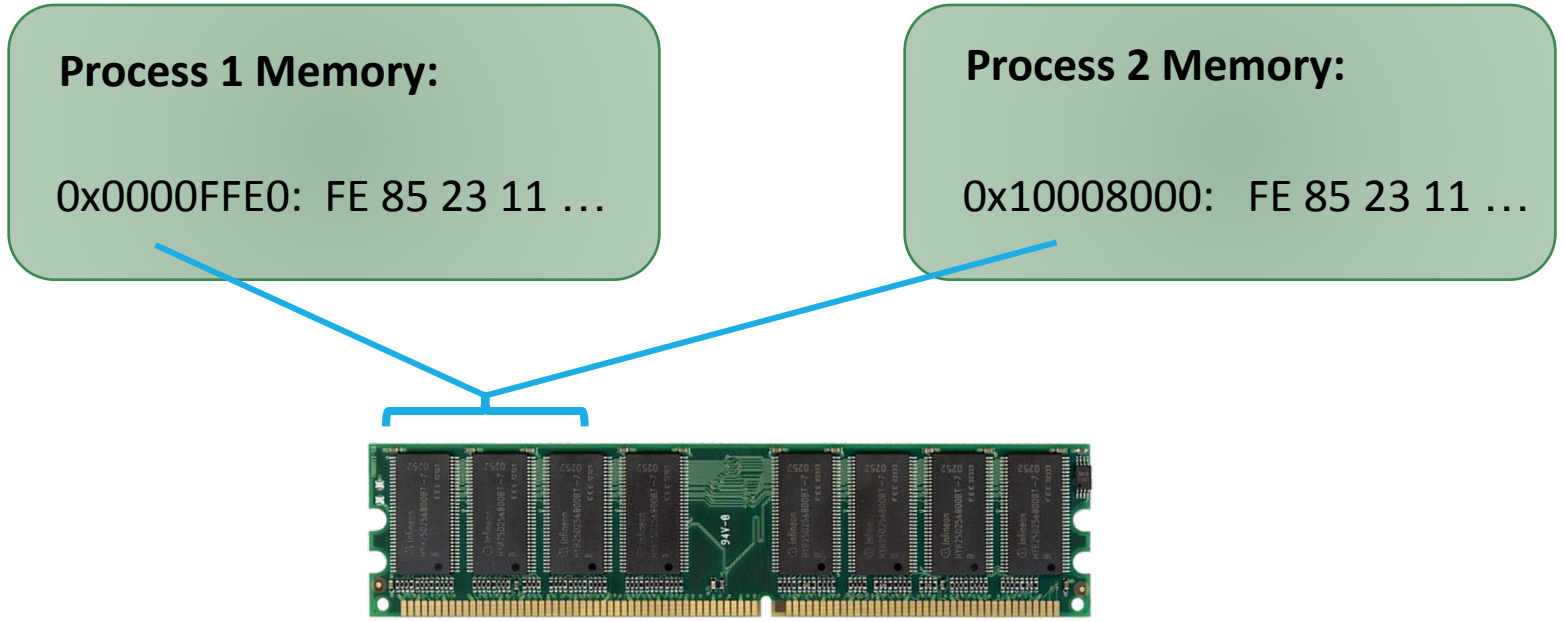
IPC: How would *YOU* do it?

Let's say two processes on the same machine DO want to share memory. How would *you* do it?

IPC: Memory Mapped Files

Memory mapped files are files whose content is mapped directly to a process' memory. For efficiency, these files usually only exist in memory (i.e. are never explicitly written to the hard disk).

The same contents are mapped to virtual address space in each process that is granted access.



IPC: Memory Mapped Files

We can create these **shared memory objects**, also sometimes simply called **shared memory** or **Memory Mapped Files**, using operating system kernel calls.

Windows, Linux, OSX:

```
Namespace:      MemoryMappedFiles
Methods:        CreateFromFile(...)
                CreateViewStream(...)
                OpenExisting(...)
```

https://docs.microsoft.com/en-us/dotnet/standard/io/memory-mapped-files

Linux/OSX:

Note: *Named* Memory mapped files are not supported on Unix for .NET

https://github.com/dotnet/runtime/issues/21863

IPC: Shared Memory Types

- **Persistent Shared Memory-Mapped Files**
  - The CreateFromFile methods create a memory-mapped file from an existing file on disk.
  - One application is to create a memory-mapped view of a part of an extremely large file and manipulates a portion of it.

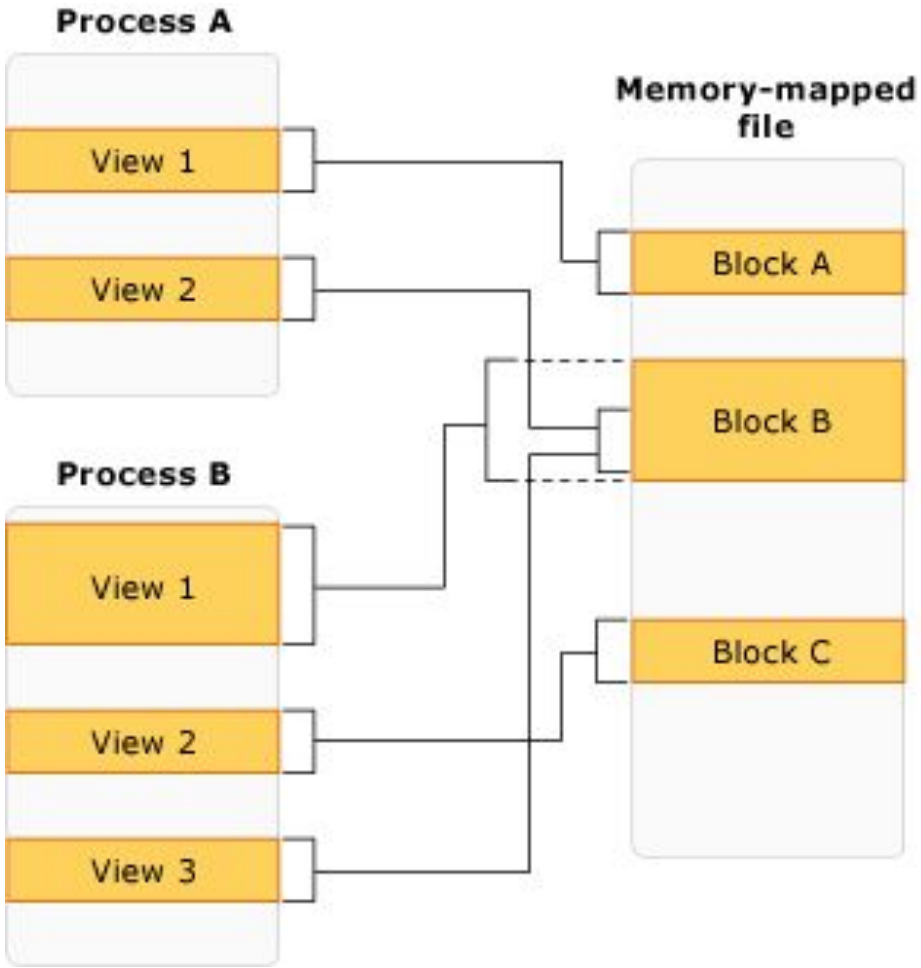
- **Non-Persistent Shared Memory-Mapped Files**
  - The CreateNew and CreateOrOpen methods create a memory-mapped file that is not mapped to an existing file on disk.
  - Commonly used for interprocess communications

IPC: View and Memory

- View:
  - create a view of the entire memory-mapped file or a part of it.
  - create multiple views to the same part of the memory-mapped file, thereby creating concurrent memory
  - There are two types of views:
    - Stream access: for IPC and non-persistent files (`MemoryMappedFile.CreateViewStream`)
    - Random access: for persistent files (`MemoryMappedFile.CreateViewAccessor`)
- Memory:
  - Memory-mapped files are accessed through the operating system's memory manager
  - The file is automatically partitioned into a number of pages and accessed as needed
  - You do not have to handle the memory management yourself.

https://docs.microsoft.com/en-us/dotnet/standard/io/memory-mapped-files

IPC: View and Memory



https://docs.microsoft.com/en-us/dotnet/standard/io/memory-mapped-files

IPC: Shared Memory (Persistent)

Rather than working directly with operating system calls, CLR provides a uniform interface:

```
// Create the memory-mapped file.
using (var mmf = MemoryMappedFile.CreateFromFile(@"c:\ExtremelyLargeImage.data", FileMode.Open, "ImgA"))
{
    // Create a random access view, from the the offset
    // to the the offset plus length
    using (var accessor = mmf.CreateViewAccessor(offset, length))
    {
        int colorSize = Marshal.SizeOf(typeof(MyColor));
        MyColor color;

        // Make changes to the view.
        for (long i = 0; i < length; i += colorSize)
        {
            accessor.Read(i, out color);
            color.Brighten(10);
            accessor.Write(i, ref color);
        }
    }
}
```

Ref

IPC: Shared Memory

```
public struct MyColor
{
    public short Red;
    public short Green;
    public short Blue;
    public short Alpha;

    // Make the view brigher.
    public void Brighten(short value)
    {
        Red = (short)Math.Min(short.MaxValue, (int)Red + value);
        Green = (short)Math.Min(short.MaxValue, (int)Green + value);
        Blue = (short)Math.Min(short.MaxValue, (int)Blue + value);
        Alpha = (short)Math.Min(short.MaxValue, (int)Alpha + value);
    }
}
```

Ref

IPC: Shared Memory

This second process can read/write from/to the shared memory as follows:

```
// Assumes another process has created the memory-mapped file.
using (var mmf = MemoryMappedFile.OpenExisting("ImgA"))
{
    using (var accessor = mmf.CreateViewAccessor(4000000, 2000000))
    {
        int colorSize = Marshal.SizeOf(typeof(MyColor));
        MyColor color;

        // Make changes to the view.
        for (long i = 0; i < 1500000; i += colorSize)
        {
            accessor.Read(i, out color);
            color.Brighten(20);
            accessor.Write(i, ref color);
        }
    }
}
```

Ref

IPC: Shared Memory (Non-Persistent)

Consider a scenario where three separate processes (console applications) that write Boolean values to a memory-mapped file. The following sequence of actions occur:

- Process A creates the memory-mapped file and writes a value to it.
- Process B opens the memory-mapped file and writes a value to it.
- Process C opens the memory-mapped file and writes a value to it.
- Process A reads and displays the values from the memory-mapped file.
- After Process A is finished with the memory-mapped file, the file is immediately reclaimed by garbage collection.

IPC: Shared Memory (Non-Persistent)

```
using (MemoryMappedFile mmf = MemoryMappedFile.CreateNew("testmap", 10000))
{
   bool mutexCreated;
   Mutex mutex = new Mutex(true, "testmapmutex", out mutexCreated);
   using (MemoryMappedViewStream stream = mmf.CreateViewStream())
   {
       BinaryWriter writer = new BinaryWriter(stream);
       writer.Write(1);
   }
   mutex.ReleaseMutex();
   Console.WriteLine("Start Process B and press ENTER to continue.");
   Console.ReadLine();
   Console.WriteLine("Start Process C and press ENTER to continue.");
   Console.ReadLine();
   mutex.WaitOne();

    ...
```

Ref

IPC: Shared Memory (Non-Persistent)

```
…
//Process A continues
using (MemoryMappedViewStream stream = mmf.CreateViewStream())
    {
        BinaryReader reader = new BinaryReader(stream);
        Console.WriteLine("Process A says: {0}", reader.ReadBoolean());
        Console.WriteLine("Process B says: {0}", reader.ReadBoolean());
        Console.WriteLine("Process C says: {0}", reader.ReadBoolean());
    }
    mutex.ReleaseMutex();
}
```

IPC: Shared Memory (Non-Persistent)

```
// Process B
using (MemoryMappedFile mmf = MemoryMappedFile.OpenExisting("testmap"))
{
    Mutex mutex = Mutex.OpenExisting("testmapmutex");
    mutex.WaitOne();

    using (MemoryMappedViewStream stream = mmf.CreateViewStream(1, 0))
    {
        BinaryWriter writer = new BinaryWriter(stream);
        writer.Write(0);
    }
    mutex.ReleaseMutex();
}
```

IPC: Shared Memory (Non-Persistent)

```
// Process C
using (MemoryMappedFile mmf = MemoryMappedFile.OpenExisting("testmap"))
{
    Mutex mutex = Mutex.OpenExisting("testmapmutex");
    mutex.WaitOne();

    using (MemoryMappedViewStream stream = mmf.CreateViewStream(2, 0))
    {
        BinaryWriter writer = new BinaryWriter(stream);
        writer.Write(1);
    }
    mutex.ReleaseMutex();
}
```

“Named” Resources

So far, we have seen two inter-process resources that we have accessed using a **name**: **mutexes**, and now **shared memory**.

This is a common way for the operating system to **connect resources** between processes. Each must supply the same name in order to connect to the same resource.

If a resource with the given name does not already exist, the operating system will create and initialize it. If it *does* exist, the OS will return the existing one.

Persistence

Process Persistence: the resource will exist as long as at least one process still has a reference to it.

Filesystem Persistence: the resource is backed by the filesystem (will persist even after reboot).

Homework

Lab 2:

- Familiarize yourself with multi threading

Reading on Memory Mapped Files:

- https://docs.microsoft.com/en-us/dotnet/standard/io/memory-mapped-files
- https://docs.microsoft.com/en-us/windows/win32/memory/file-mapping