Game Engine Architecture

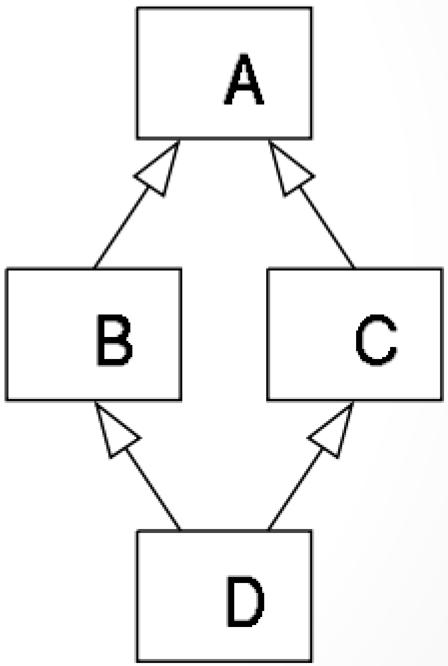
Chapter 3

Fundamentals of Software Engineering for Games

Hooman Salamat

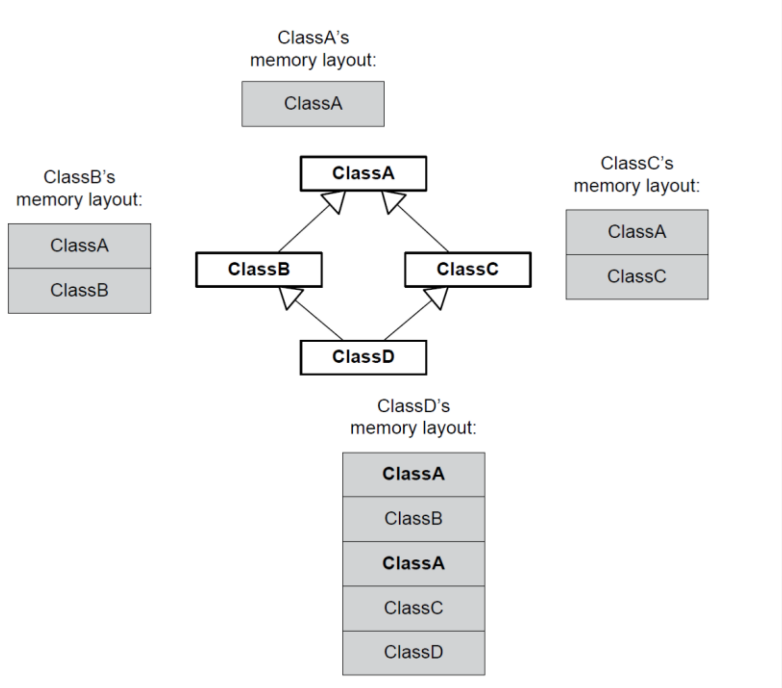
Brief Review of Object-Oriented Programming

- Classes and Objects
  - A *class* is a collection of attributes (data) and behaviors (code)
- Encapsulation
  - an object presents only a limited interface to the outside world
- Inheritance
  - allows new classes to be defined as *extensions* to preexisting classes
  - Inheritance creates an "is-a" relationship between classes.
  - For example, a circle *is a* type of shape.
  - We can draw diagrams of class hierarchies using the conventions defined by the Unified Modeling Language (UML).
- *Multiple Inheritance (MI)*
  - *deadly diamond:* inheritance diagram is in the shape of a diamond

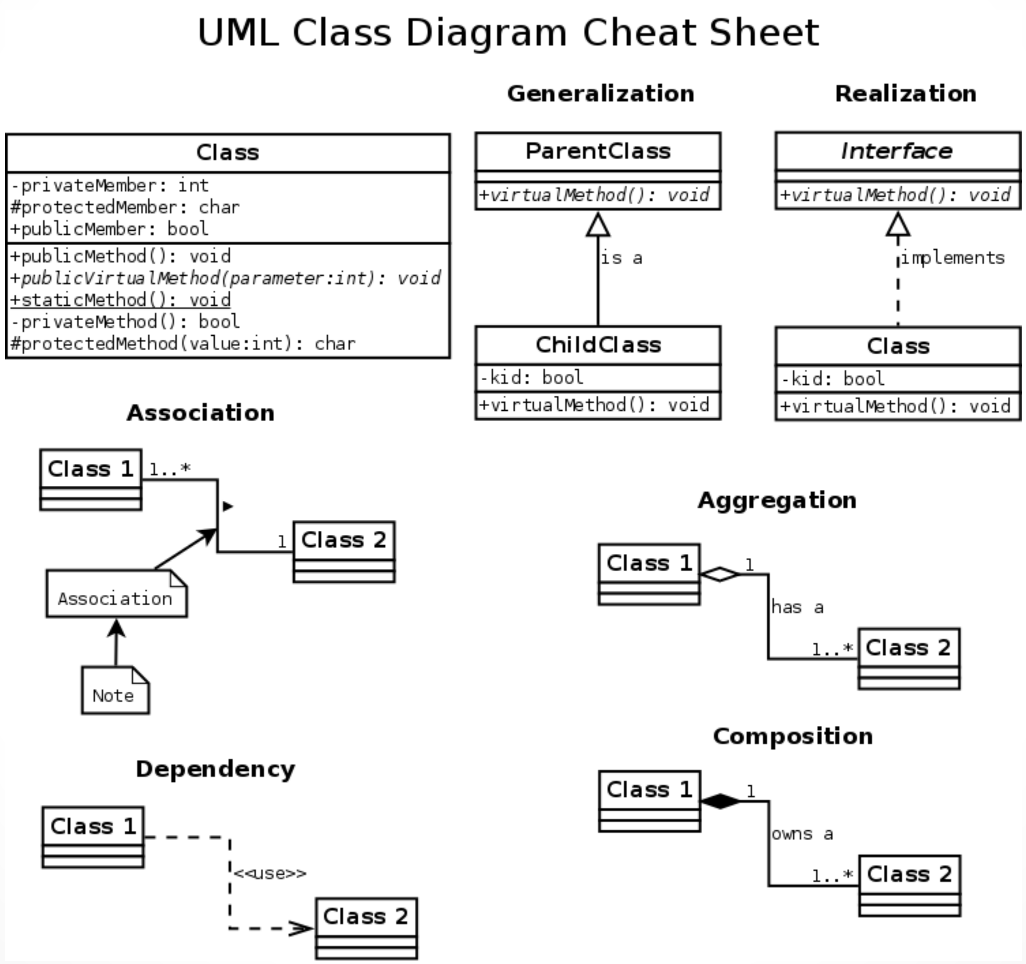


Deadly Diamond

- a derived class ends up containing *two copies* of a grandparent base class
- Multiple inheritance also complicates casting, because the actual address of a pointer may change depending on which base class it is cast to.
- In C++, *virtual inheritance* avoid this doubling of the grandparent's data.

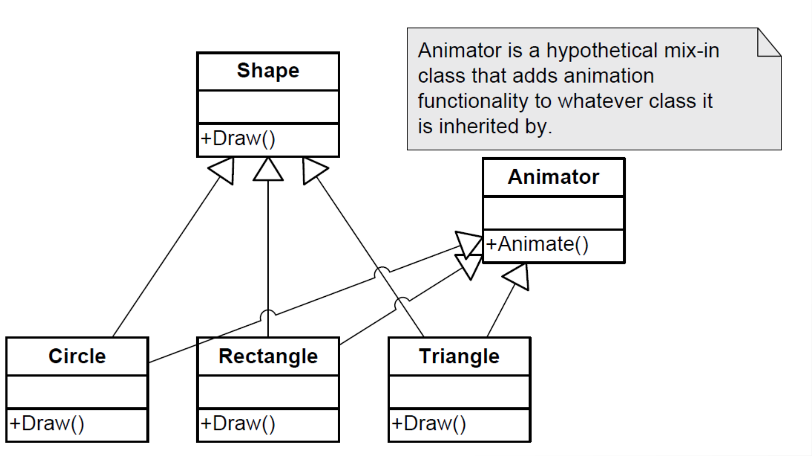


UML


UML Class Diagram Cheat Sheet
Generalization
Realization
Class
-privateMember: int
#protectedMember: char
+publicMember: bool
+publicMethod(): void
+publicVirtualMethod(parameter:int): void
+staticMethod(): void
-privateMethod(): bool
#protectedMethod(value:int): char
ParentClass
+virtualMethod(): void
is a
ChildClass
-kid: bool
+virtualMethod(): void
Interface
+virtualMethod(): void
implements
Class
-kid: bool
+virtualMethod(): void
Association
Class 1
1..*
1
Class 2
Association
Note
Aggregation
Class 1
1
has a
1..*
Class 2
Composition
Class 1
1
owns a
1..*
Class 2
Dependency
Class 1
<<use>>
Class 2

Mix-in classes

- contains methods for use by other classes without having to be the parent class of those other classes.



Polymorphism

- allows a collection of objects of different types to be manipulated through a single *common interface*
- One way to draw this heterogeneous collection of shapes is to use a switch statement
- drawShapes() function needs to "know" about all of the kinds of shapes that can be drawn.
- difficult to add new types of shapes to the system

```
void drawShapes(std::list<Shape*>& shapes)
{
std::list<Shape*>::iterator pShape = shapes.begin();
std::list<Shape*>::iterator pEnd = shapes.end();
for ( ; pShape != pEnd; pShape++)
{
switch (pShape->mType)
{
case CIRCLE:
// draw shape as a circle
break;
case RECTANGLE:
// draw shape as a rectangle
break;
case TRIANGLE:
// draw shape as a triangle
break;
//...
}
}
}
```

Polymorphism

- The solution is to insulate the majority of our code from any knowledge of the types of objects with which it might be dealing.
- A *virtual function*—
- the C++ language's primary polymorphism mechanism

```
struct Shape
{
virtual void Draw() = 0; // pure virtual function
virtual ~Shape() { } // ensure derived dtors are virtual
};
struct Circle : public Shape
{
virtual void Draw()
{
// draw shape as a circle
}
};
struct Rectangle : public Shape
{
virtual void Draw()
{
// draw shape as a rectangle
}
};
struct Triangle : public Shape
{
virtual void Draw()
{
// draw shape as a triangle
}
};
```

Composition and Aggregation

- *Composition* is the practice of using a *group* of *interacting* objects to accomplish a high-level task.
- object composition is the process of creating complex objects from simpler ones.
- the “has-a” relationship is called *composition*.
- spaceship *has an* engine, which in turn *has a* fuel tank.

- an aggregation is still a part-whole relationship, where the parts are contained within the whole, and it is a unidirectional relationship
- parts can belong to more than one object at a time, and the whole object is not responsible for the existence and lifespan of the parts.
- the “uses-a” relationship is called *aggregation*
- every person has an address. However, that address can belong to more than one person at a time

Design Patterns

The most well-known book on this topic is probably the "Gang of Four" book
https://en.wikipedia.org/wiki/Design_Patterns

- *Singleton*. This pattern ensures that a particular class has only one instance (the *singleton instance*) and provides a global point of access to it.
- *Iterator*. An iterator provides an efficient means of accessing the individual elements of a collection, without exposing the collection's underlying
- *Abstract factory*. An abstract factory provides an interface for creating families of related or dependent classes without specifying their concrete classes.

RAII Design Pattern

- “resource acquisition is initialization” pattern (RAII).
- the acquisition and release of a resource (such as a file, a block of dynamically allocated memory, or a mutex lock) are bound to the constructor and destructor of a class

```
void bad()
{
    m.lock(); // acquire the mutex
    f();
// if f() throws an exception, the mutex is never
released
    if (!everything_ok()) return;
// early return, the mutex is never released
    m.unlock();
// if bad() reaches this statement, the mutex is
released
}

void good()
{
    std::lock_guard<std::mutex> lk(m);
// RAII class: mutex acquisition is initialization
    f();
// if f() throws an exception, the mutex is
released
    if (!everything_ok()) return;
// early return, the mutex is released
}
// if good() returns normally, the mutex is
released
```

C++ Language Standardization

- C++*98* was the first official C++ standard, established by the ISO in 1998.
- C++*03* was introduced in 2003
- C++11 added a large number of powerful new features to the language
- C++*14* was approved by the ISO on August 18, 2014
- C++*17* was published by the ISO on July 31, 2017
- **C++20 is supposed to be published by the end of 2020**

Cost of Switching between Standards

- decide on the most advanced C++ standard to support, and then stick with it for a reasonable length of time
- remember that when you have a hammer, everything can tend to look like a nail. Don't be tempted to use features of your language just because they're there (or because they're new).
- A judicious and carefully considered approach will result in a stable codebase that's as easy as possible to understand, reason about, debug and maintain.

Data, Code and Memory Layout

- People think most naturally in *base ten*, also known as *decimal notation*
- integers and real valued numbers need to be stored in the computer's memory (*base-two* representation)
- Computer scientists sometimes use a prefix of "0b" to represent binary numbers
- 0b1101 = (1 23) + (1 22) + (0 21) + (1 20) = 8 + 4 + 0 + 1 = 13.
- Another common notation popular in computing circles is *hexadecimal*, or *base 16*.
- A prefix of "0x" is used to denote hex numbers in the C and C++

Signed and Unsigned Integers

- To represent a *signed* integer in 32 bits, the *sign and magnitude* encoding reserves the most significant bit as a *sign bit*.
- When this bit is zero, the value is positive, and when it is one, the value is negative.
- Most microprocessors use *two's complement* notation
  - So values from 0x00000000 (0) to 0x7FFFFFFF (2,147,483,647) represent positive integers, and 0x80000000 (-2,147,483,648) to 0xFFFFFFFF (-1) represent negative integers.

Fixed-Point Notation

- To represent fractions and irrational numbers we need a different format that expresses the concept of a decimal point.
- how many bits will be used to represent the whole part of the number, and the rest of the bits are used to represent the fractional part.
- As we move from left to right (i.e., from the most significant bit to the least significant bit), the magnitude bits represent decreasing powers of two (…, 16, 8, 4, 2, 1), while the fractional bits represent decreasing *inverse* powers of two (1/2, 1/4, 1/8, 1/16 , . . .)
- to store the Number -173.25 in 32-bit fixed-point notation with one sign bit, 16 bits for the magnitude and 15 bits for the fraction: 0x8056A000

1	0	0	0	0	0	0	0	0	1	0	1	0	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	= –173.25
31																15															0	

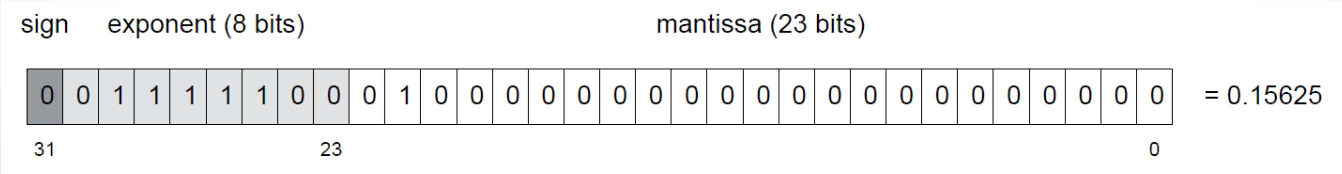
0x80 0x56 0xA0 0x00

Fixed-Point Notation

- The problem with fixed-point notation is that it constrains both the range of magnitudes that can be represented and the amount of precision we can achieve in the fractional part.
- Consider a 32-bit fixed-point value with 16 bits for the magnitude, 15 bits for the fraction and a sign bit.
- This format can only represent magnitudes up to 65,535, which isn't particularly large.
- To overcome this problem, we employ a *floating-point* representation.

Floating-Point Notation

- the position of the decimal place is arbitrary and is specified with the help of an exponent
- A floating-point number is broken
- into three parts:
    - the *mantissa*, which contains the relevant digits of the number on both sides of the decimal point,
    - the *exponent*, which indicates where in that string of digits the decimal point lies,
    - and a *sign bit*
- IEEE-754 states that a 32-bit floating-point number will be represented with the sign in the most significant bit, followed by 8 bits of exponent and
- finally 23 bits of mantissa.



Floating-Point Notation

- $s = 0$
- $e = 0b01111100 = 124$
- and $m = 0b0100... = 1/4$

$$
\begin{aligned}
v &= s \times 2^{(e-127)} \times (1+m) \\
&= (+1) \times 2^{(124-127)} \times (1+\tfrac{1}{4}) \\
&= 2^{-3} \times \tfrac{5}{4} \\
&= \tfrac{1}{8} \times \tfrac{5}{4} \\
&= 0.125 \times 1.25 = 0.15625.
\end{aligned}
$$

- The value v represented by a sign bit s, an exponent e and a mantissa m is
- $v = s * 2^{(e-127)} * (1 + m)$.
- The sign bit s has the value +1 or -1.
- The exponent e is biased by 127 so that negative exponents can be easily represented.
- The mantissa begins with an implicit 1 that is not actually stored in memory, and the rest of the bits are interpreted as inverse powers of two.
- Hence the value represented is really 1

Ogre Primitive Data Types

```
// Integer formats of fixed bit width
typedef unsigned int uint32;
typedef unsigned short uint16;
typedef unsigned char uint8;
typedef int int32;
typedef short int16;
typedef signed char int8;
// define uint64 type
#if OGRE_COMPILER == OGRE_COMPILER_MSVC
    typedef unsigned __int64 uint64;
    typedef __int64 int64;
#else
    typedef unsigned long long uint64;
    typedef long long int64;
#endif

#if OGRE_DOUBLE_PRECISION == 1
    typedef double Real;
#else
     typedef float Real;
#endif
```

- Ogre::uint8, Ogre::uint16 and Ogre::uint32 are the basic unsigned sized integral types.
- Ogre::Real defines a real floating-point value.

OgreMath.h

- Ogre::Radian and Ogre::Degree are wrapper classes (around Ogre::Real) which indicates a given angle value is in Radians or Degree.
- Radian values are interchangeable with Degree values, and conversions will be done automatically between them.
- Ogre::Angle represents an angle in the current "default" angle unit. The programmer can define whether the default will be radians or degrees when the OGRE application first starts up.

Multibyte Values

- Values that are larger than eight bits (one byte) wide are called *multibyte quantities*.
- For example, the integer value 4660 = 0x1234 is represented by the two bytes 0x12 and 0x34.
- 0x12 the most significant byte and 0x34 the least significant byte
- In a 32-bit value, such as 0xABCD1234, the most-significant byte is 0xAB and the least significant is 0x34.
- The same concepts apply to 64-bit integers and to 32- and
- 64-bit floating-point values as well.

Endianness

- Multibyte integers can be stored into memory in one of two ways:
- *Little-endian:* If a microprocessor stores the least significant byte of a multibyte value at a lower memory address than the most significant byte
- *Big-endian:* stores the most significant byte of a multibyte value at a lower memory address

```
U32 value = 0xABCD1234;
U8* pBytes = (U8*)&value;
```

Big-endian		Little-endian	
pBytes + 0x0	0xAB	pBytes + 0x0	0x34
pBytes + 0x1	0xCD	pBytes + 0x1	0x12
pBytes + 0x2	0x12	pBytes + 0x2	0xCD
pBytes + 0x3	0x34	pBytes + 0x3	0xAB

Big- and little-endian representations of the value 0xABCD1234.

Why Endianness Matters?

- games are usually *developed* on a Windows or Linux machine running an Intel Pentium processor (which is little-endian),
- but *run* on a console such as the Wii, Xbox or PlayStation —all three of which utilize a variant of the PowerPC processor (which can be configured to use either endianness, but is big-endian by default
- Solution:
  - You could write all your data files as text and store all multibyte numbers as sequences of decimal or hexadecimal digits, one character (one byte) per digit. This would be an inefficient use of disk space, but it would work.
  - You can have your tools endian-swap the data prior to writing it into a binary data file.

Integer Endian Swapping

```
struct Example
{
U32 m_a;
U16 m_b;
U32 m_c;
};

inline U16 swapU16(U16 value)
{
return ((value & 0x00FF) << 8)
| ((value & 0xFF00) >> 8);
}
inline U32 swapU32(U32 value)
{
return ((value & 0x000000FF) << 24)
| ((value & 0x0000FF00) << 8)
| ((value & 0x00FF0000) >> 8)
| ((value & 0xFF000000) >> 24);
}
```

Floatingpoint Endian Swapping

- an IEEE - 754 floating - point value has a detailed internal structure involving some bits for the mantissa, some bits for the exponent and a sign bit.
- However, you can endian - swap it just as if it were an integer, because bytes are bytes.

```
union U32F32
{
U32 m_asU32;
F32 m_asF32;
};
inline F32 swapF32(F32 value)
{
U32F32 u;
u.m_asF32 = value;
// endian-swap as integer
u.m_asU32 = swapU32(u.m_asU32);
return u.m_asF32;
}
```

Kilobytes versus Kibibytes

Metric (SI)			IEC		
Value	**Unit**	**Name**	**Value**	**Unit**	**Name**
1000	kB	kilobyte	1024	KiB	kibibyte
1000^2	MB	megabyte	1024^2	MiB	mebibyte
1000^3	GB	gigabyte	1024^3	GiB	gibibyte
1000^4	TB	terabyte	1024^4	TiB	tebibyte
1000^5	PB	petabyte	1024^5	PiB	pebibyte
1000^6	EB	exabyte	1024^6	EiB	exbibyte
1000^7	ZB	zettabyte	1024^7	ZiB	zebibyte
1000^8	YB	yottabyte	1024^8	YiB	yobibyte

- When we say "kilobyte," we usually means 1024 bytes.
- (The International System of Units) SI units define the prefix "kilo" to mean 10^3 or 1000, not 1024.
- To resolve this ambiguity, the International Electrotechnical Commission (IEC) in 1998 established a new set of SI-like prefixes for use in computer science.

Declarations, Definitions and Linkage

- The compiler translates one .cpp file at a time, and for each one it generates an output file called an object file (.o or .obj).
- A .cpp file is the smallest unit of translation operated on by the compiler: “translation unit”
- An object file contains not only the compiled machine code for all of the functions defined in the .cpp file, but also all of its global and static variables.
- An object file may contain *unresolved references* to functions and global variables defined in *other* .cpp files.
- Linker’s job to combine all of the object files into a final

executable image.

Definitions in Header Files and Inlining

- if a header file containing a *definition* is #included
- into more than one .cpp file → "multiply defined symbol" linker error.
- Inline function definitions are an exception to this rule, because each invocation of an inline function gives rise to a brand new copy of that function's machine code

inline

- The inline keyword is really just a hint to the compiler.
- Note that it is *not* sufficient to tag a function *declaration* with the inline keyword in a .h file and then place the body of that function in a .cpp file.
- The compiler must be able to "see" the body of the function in order to inline it.
- inline function definitions *must* be placed in header files if they are to be used in more than one translation unit.

foo.h

```
// This function definition will be inlined properly.
inline int max(int a, int b)
{
return (a > b) ? a : b;
}

// This declaration cannot be inlined because the
// compiler cannot "see" the body of the function.
inline int min(int a, int b);
```

foo.cpp

```
// The body of min() is effectively "hidden" from the
// compiler, so it can ONLY be inlined within foo.cpp.
int min(int a, int b)
{
return (a <= b) ? a : b;
}
```

So why inline?

- Compiler does a cost/ benefit analysis of each inline function, weighing the size of the function's code versus the potential performance benefits of inling it,
- the compiler gets the final say as to whether the function will really be inlined or not.
- Some compilers provide syntax like __forceinline, allowing the programmer to bypass the compiler's cost/benefit analysis and control function inlining directly.
- Inline functions provide following advantages:

  1) **Function call overhead** doesn't **occur**.

  2) It also saves the overhead of push/**pop variables** on the stack when function is called.

  3) It also saves overhead of a return call from a function.

Linkage

- Every definition in C and C++ has a property known as *linkage*.
- Technically speaking, *declarations* don't have a linkage property at all, because they do not allocate any storage in the executable image
- A definition with *external linkage* is visible to and can be referenced by translation units other than the one in which it appears.
- A definition with *internal linkage* can only be "seen" inside the translation unit
- linkage is the translation unit's equivalent of the public: and private: keywords in C++ class definitions.
- By default, definitions have external linkage. The static keyword is used to change a definition's linkage to internal.

static definitions

foo.cpp

```
// This variable can be used by other .cpp files (external
linkage).
U32 gExternalVariable;

// This variable is only usable within foo.cpp (internal  linkage).

static U32 gInternalVariable;

// This function can be called from other .cpp files  (external
linkage).
void externalFunction()
{
// ...
}

// This function can only be called from within foo.cpp
// (internal linkage).
static void internalFunction()
{
// ...
}
```

- two or more identical static definitions in two or more different .cpp files are considered to be *distinct entities* by the linker

Extern vs. static

bar.cpp

```
// This declaration grants access to foo.cpp's variable.
extern U32 gExternalVariable;

// This 'gInternalVariable' is distinct from the one
// defined in foo.cpp -- no error. We could just as
// well have named it gInternalVariableForBarCpp -- the
// net effect is the same.
static U32 gInternalVariable;

// This function is distinct from foo.cpp's
// version -- no error. It acts as if we had named it
// internalFunctionForBarCpp().
static void internalFunction()
{
// ...
}

// ERROR -- multiply defined symbol!
void externalFunction()
{
// ...
}
```

Memory Layout of a C/C++ Program

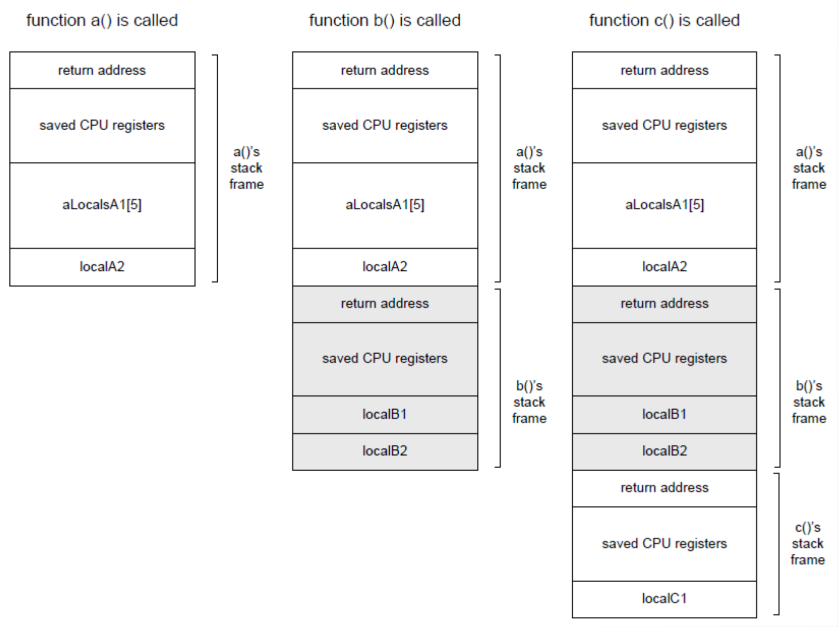
- Executable Image: *executable and linking format* (ELF) in Unix-Like OS and .exe in Windows.
- The executable file always contains a partial *image* of the program as it will exist in memory when it runs.
- The executable image is divided into contiguous blocks called *segments* or *sections*.
    - *Text segment*. Sometimes called the *code segment*
    - *Data segment*. contains all *initialized* global and static variables.
    - *BSS segment*: "block started by symbol" segment contains *uninitialized* global and static variables
    - *Read-only data segment: rodata* segment contains any read-only (constant) global data

Program Stack

- When an executable program is loaded into memory and run, OS reserves an area of memory for the *program stack*.
- Whenever a function is called, a contiguous area of stack memory is pushed onto the stack—*stack frame*.
- If function a() calls another function b(), a new stack frame for b() is pushed on top of a()'s frame. When b() returns, its stack frame is popped,
- Stack frames stores
  - the *return address* of the calling function
  - The contents of all relevant *CPU registers*
  - all *local variables* declared by the function: *automatic variables*.

Stack frame

```
void c()
{
  U32 localC1;
  // ...
}

F32 b()
{
  F32 localB1;
  I32 localB2;
  // ...
  c();
  // ...
  return localB1;
}

void a()
{
U32 aLocalsA1[5];
// ...
F32 localA2 = b();
// ...
}
```



Dynamic Allocation Heap

- To allow for dynamic allocation, the operating system maintains a block of memory (Heap Memory) for each running process
- In C++, the global new and delete operators are used to allocate and free memory to and from the free store.

Member Variables

- C structs and C++ classes allow variables to be grouped into logical units.
- class or struct *declaration* allocates no memory.
- data—a cookie cutter which can be used to stamp out *instances* of that struct or class later on

```
struct Foo // struct declaration
{
U32 mUnsignedValue;
F32 mFloatValue;
bool mBooleanValue;
};
```

Allocation

- Once a struct or class has been declared, it can be allocated (defined) in any of the ways that a primitive data type can be allocated;
  - as an automatic variable, on the program stack;

```
void someFunction()
{
Foo localFoo;
// ...
}
```

  - as a global, file-static or function-static;

```
Foo gFoo;
static Foo sFoo;
void someFunction()
{
static Foo sLocalFoo;
// ...
}
```

Dynamic Allocation

- a struct or class can be also dynamically allocated from the heap.
- the pointer or reference variable used to hold the address of the data can itself be allocated as an automatic, global, static or even dynamically.

```
Foo* gpFoo = nullptr; // global pointer to a Foo
void someFunction()
{
// allocate a Foo instance from the heap
gpFoo = new Foo;
// ...
// allocate another Foo, assign to local pointer
Foo* pAnotherFoo = new Foo;
// ...
// allocate a POINTER to a Foo from the heap
Foo** ppFoo = new Foo*;
(*ppFoo) = pAnotherFoo;
}
```

Class-Static Members

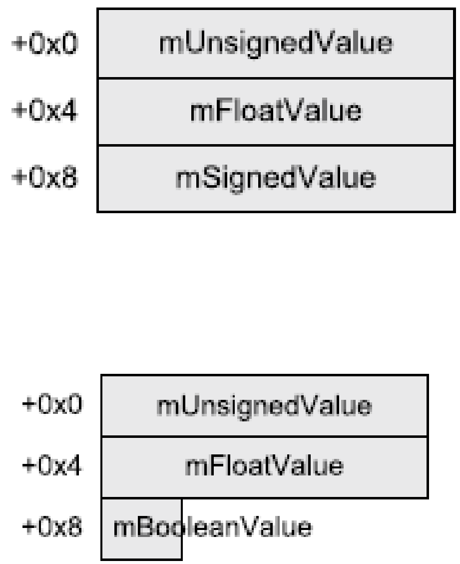
- When used at file scope, static means "restrict the visibility of this variable or function so it can only be seen inside this .cpp file."
- When used at function scope, static means "this variable is a global, not an automatic, but it can only be seen inside this function."
- When used inside a struct or class declaration, static means "this variable is not a regular member variable, but instead acts just like a global."

Static Members

```
//foo.h
class Foo
{
public:
static F32 sClassStatic; // allocates no memory!
};

//foo.cpp
static Foo sFoo; // restrict the visibility
F32 Foo::sClassStatic = -1.0f; // define memory and initialize
void someFunction()
{
static Foo sLocalFoo; // acts just like a global
// ...
}
```

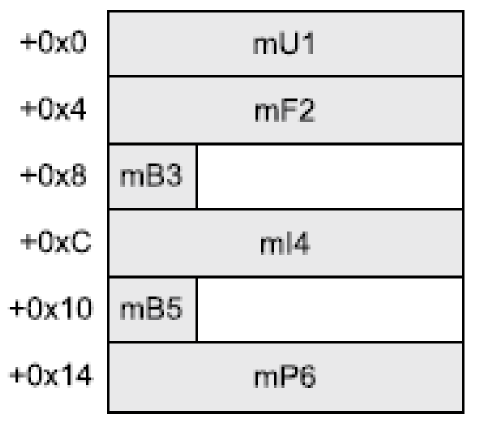
Object Layout in Memory

struct Foo
{
U32 mUnsignedValue;
F32 mFloatValue;
I32 mSignedValue;
};
struct Bar
{
U32 mUnsignedValue;
F32 mFloatValue;
bool mBooleanValue; // diagram assumes this is 8 bits
};



Alignment and Packing

```
struct InefficientPacking
{
U32 mU1; // 32 bits
F32 mF2; // 32 bits
U8 mB3; // 8 bits
I32 mI4; // 32 bits
bool mB5; // 8 bits
char* mP6; // 32 bits
};
```



Alignment

- Why does the compiler leave these “holes”?
- every data type has a natural *alignment*, which must be respected in order to permit the CPU to read and write memory effectively
- The *alignment* of a data
- object refers to whether its *address in memory* is a multiple of its *size* (which is generally a power of two):

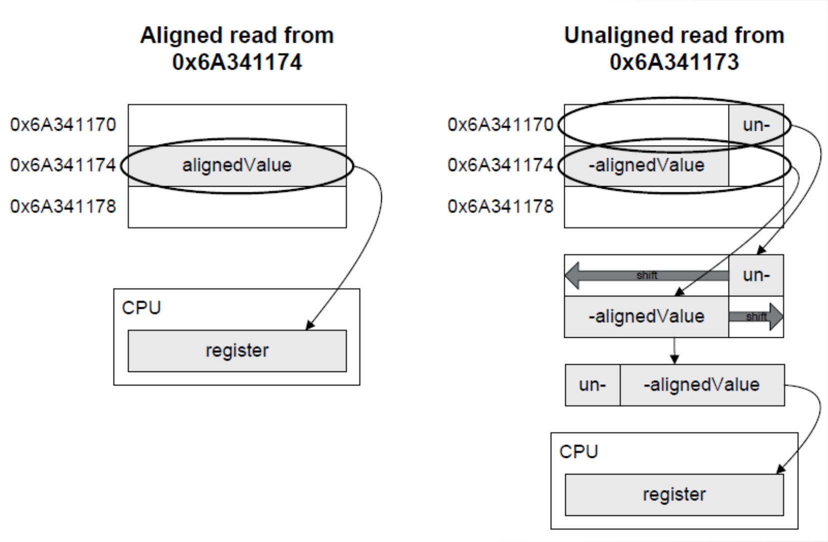
- An object with 1-byte alignment resides at any memory address.
- An object with 2-byte alignment resides only at even addresses (i.e., addresses whose least significant nibble is 0x0, 0x2, 0x4, 0x8, 0xA, 0xC or 0xE).
- An object with 4-byte alignment resides only at addresses that are a multiple of four (i.e., addresses whose least significant nibble is 0x0, 0x4, 0x8 or 0xC).
- A 16-byte aligned object resides only at addresses that are a multiple of 16 (i.e., addresses whose least significant nibble is 0x0).

Alignment Example

- if a program requests that a 32-bit (4-byte) integer be read from address 0x6A341174,
- the memory controller will load the data happily because the address is 4-byte aligned
- in this case, its least significant nibble is 0x4.

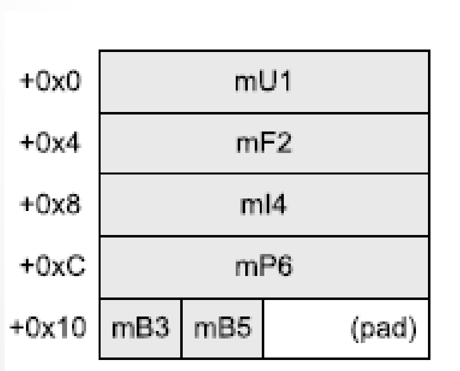
- if a request is made to load a 32-bit integer from address 0x6A341173,
- the memory controller now has to read *two* 4-byte blocks: the one at 0x6A341170 and the one at 0x6A341174

- It must then mask and shift the two parts of the 32-bit integer and logically OR them together into the destination register on the CPU.



Packing

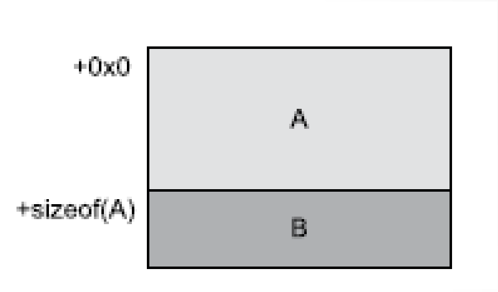
struct MoreEfficientPacking
{
U32 mU1; // 32 bits (4-byte aligned)
F32 mF2; // 32 bits (4-byte aligned)
I32 mI4; // 32 bits (4-byte aligned)
char* mP6; // 32 bits (4-byte aligned)
U8 mB3; // 8 bits (1-byte aligned)
bool mB5; // 8 bits (1-byte aligned)
};



struct BestPacking
{
U32 mU1; // 32 bits (4-byte aligned)
F32 mF2; // 32 bits (4-byte aligned)
I32 mI4; // 32 bits (4-byte aligned)
char* mP6; // 32 bits (4-byte aligned)
U8 mB3; // 8 bits (1-byte aligned)
bool mB5; // 8 bits (1-byte aligned)
U8 _pad[**2**]; // **explicit padding**
};

Memory Layout of C++ Classes

- Two things make C++ classes a little different from C structures in terms of memory layout: *inheritance* and *virtual functions*.
- When class B inherits from class A, B's data members simply appear immediately after A's in memory
- Each new derived class simply tacks its data members on at the end

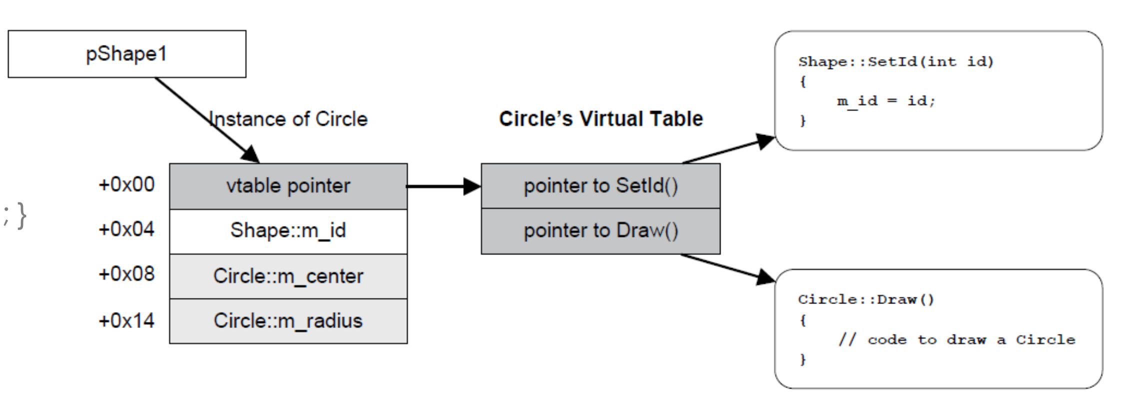


virtual table pointer

- If a class contains or inherits one or more *virtual functions*, then four additional bytes (or eight bytes if the target hardware uses 64-bit addresses) are added to the class layout
- These four or eight bytes are collectively called the *virtual table pointer*
- they contain a pointer to a data structure known as the *virtual function table* or *vtable*
- The vtable for a particular class contains pointers to all the virtual functions that it declares or inherits.
- Each concrete class has its own virtual table, and every instance of that class has a pointer to it, stored in its vpointer.

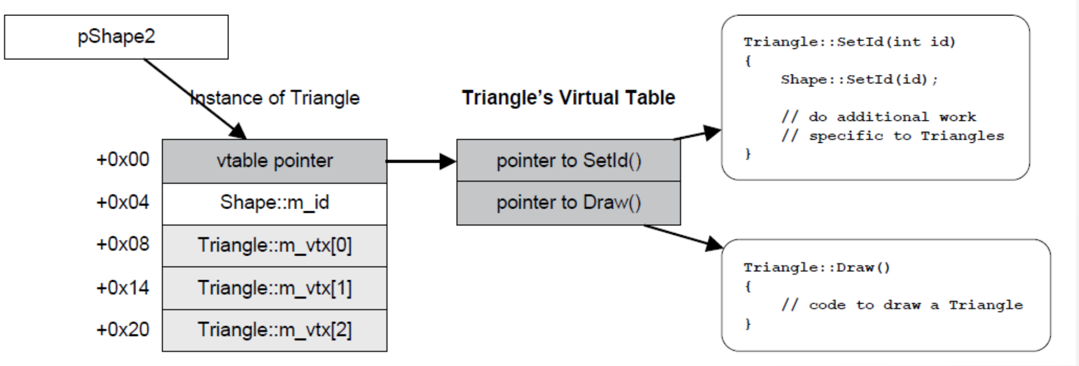
Vtable pointer example

```
class Shape
{
public:
virtual void SetId(int id) { m_id = id; }
int GetId() const { return m_id; }
virtual void Draw() = 0; // pure virtual -- no impl.
private:
int m_id;
};
class Circle : public Shape
{
public:
void SetCenter(const Vector3& c) { m_center=c; }
Vector3 GetCenter() const { return m_center; }
void SetRadius(float r) { m_radius = r; }
float GetRadius() const { return m_radius; }
virtual void Draw()
{
// code to draw a circle
}
private:
Vector3 m_center;
float m_radius;
};
```

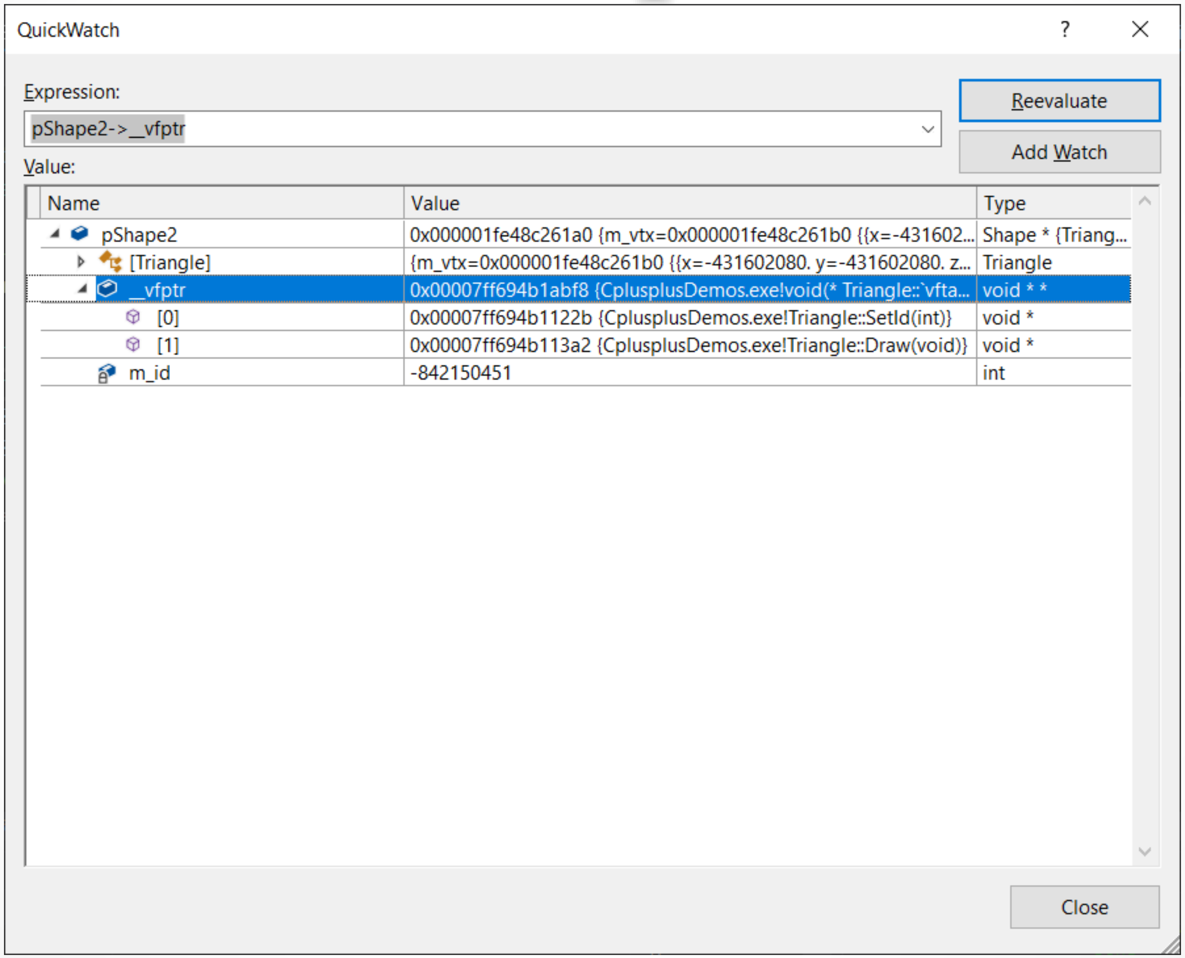


Vtable pointer example

```
class Triangle : public Shape
{
public:
void SetVertex(int i, const Vector3& v);
Vector3 GetVertex(int i) const { return m_vtx[i]; }
virtual void Draw()
{
// code to draw a triangle
}
virtual void SetId(int id)
{
// call base class' implementation
Shape::SetId(id);
// do additional work specific to Triangles...
}
private:
Vector3 m_vtx[3];
};
// ----------------------------
void main(int, char**)
{
Shape* pShape1 = new Circle;
Shape* pShape2 = new Triangle;
pShape1->Draw();
pShape2->Draw();
// ...
}
```

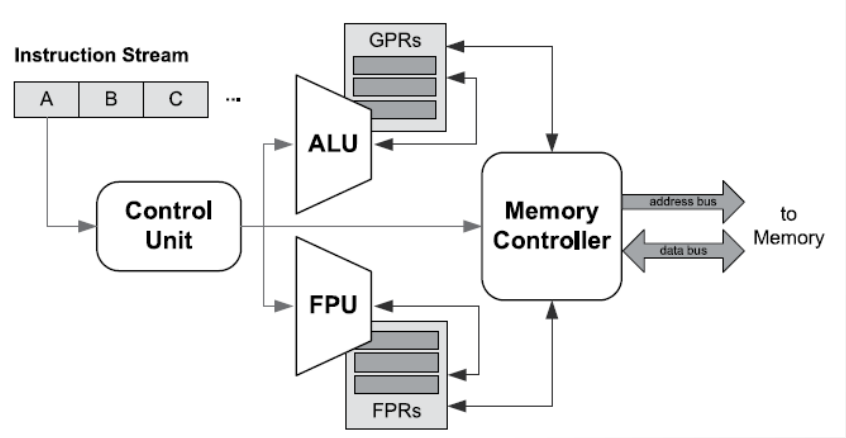


__vfptr


QuickWatch
?
Expression:
pShape2->__vfptr
Reevaluate
Add Watch
Value:
Name
Value
Type
pShape2
0x000001fe48c261a0 {m_vtx=0x000001fe48c261b0 {{x=-431602...
Shape * {Triang...
[Triangle]
{m_vtx=0x000001fe48c261b0 {{x=-431602080. y=-431602080. z...
Triangle
__vfptr
0x00007ff694b1abf8 {CplusplusDemos.exe!void(* Triangle::`vfta...
void * *
[0]
0x00007ff694b1122b {CplusplusDemos.exe!Triangle::SetId(int)}
void *
[1]
0x00007ff694b113a2 {CplusplusDemos.exe!Triangle::Draw(void)}
void *
m_id
-842150451
int
Close

Computer Hardware Fundamentals

- CPU
  - an *arithmetic/logic unit* (ALU) for performing integer arithmetic and bit shifting,
  - a *floating-point unit* (FPU) for doing floating-point arithmetic (typically using the IEEE 754 floating-point standard representation),
  - all modern CPUs also contain a *vector processing unit* (VPU)
    which is capable of performing floating-point and integer operations on multiple data items in parallel
  - a *memory controller* (MC) or *memory management unit* (MMU) for interfacing
    with on-chip and off-chip memory devices,
  - a bank of *registers* which act as temporary storage during calculations
  - a *control unit* (CU) for decoding and dispatching machine language instructions
    to the other components on the chip, and routing data between
    them.
- All of these components are driven by a periodic square wave signal known as the *clock*.



Registers

- an ALU or FPU can usually only perform calculations on data that exists in special high-speed memory cells called *registers*.
- They're usually implemented using fast, high-cost multi-ported static RAM or SRAM.
- A bank of registers within a CPU is called a *register file*.
- they typically don't have addresses but they do have names. These could be as simple as R0, R1, R2

Registers

- Some of the registers in a CPU are designed to be used for general calculations.
- They're appropriately named *general-purpose registers* (GPR).
- Every CPU also contains a number of *special-purpose registers* (SPR). These include:
  - the *instruction pointer* (IP)-> the address of the currently-executing instruction
  - the *stack pointer* (SP)-> the address of the top of the program's call stack
  - the *base pointer* (BP)-> contains the base address of the current function's *stack frame* on the call stack.
  - the *status register* -> the results of the most-recent ALU operation.

Clock Speed versus Processing

- The "processing power" of a CPU or computer can be defined in various ways.
- One common measure is the *throughput* of the machine—the number of operations it can perform during a given interval of time.
- Throughput is expressed either in units of millions of instructions per second (MIPS) or floating-point
- operations per second (FLOPS)
- Because instructions or floating-point operations don't generally complete in exactly one

Memory

- *read-only memory* (ROM)
- *Electronically erasable programmable ROM* or EEPROM can be reprogrammed over and over
  - Flash drives are one example of EEPROM
- read/write memory, or *random access memory* (RAM)
  - RAM can be further divided into *static* RAM (SRAM) and *dynamic* RAM
  - RAM retain their data as long as power is applied to them.
  - But unlike static RAM, dynamic RAM also needs to be "refreshed" periodically (by reading the data and then re-writing it) in order to prevent its contents from disappearing

Instruction Set Architecture (ISA)

- The set of all instructions supported by a given CPU, its addressing modes and the in-memory instruction format, is called its *instruction set architecture* or ISA.
- the following categories of instruction types are common to pretty much every ISA:
    - *Move (Load & Store)*
    - *Arithmetic operations:* addition, subtraction, multiplication and division
    - *Bitwise operators:* AND, OR, XOR
    - *Shift/rotate operators* (bits rolling off one end)
    - *Comparison*
    - *Jump and branch*
    - *Push and pop*
    - *Function call and return*
    - *Interrupts* -> such as an input becoming available

Machine Language

- Every machine language instruction is comprised of three basic parts:
  - an *opcode*, which tells the CPU which operation to perform
  - zero or more *operands* which specify the inputs and/or outputs of the instruction
  - some kind of *options field*, specifying things like the *addressing mode* of the instruction
- In some ISAs, all instructions occupy a fixed number of bits; this is typical of *reduced instruction set computers* (RISC).
- In other ISAs, different types of instructions may be encoded into differently-sized instruction words; this is common in *complex instruction set computers* (CISC).

Instruction word

- The opcode and operands (if any) of an ML instruction are packed into a contiguous sequence of bits called an *instruction word*.
- Instruction words are often multiples of 32 or 64 bits, because this matches the width of the CPU's registers and/or data bus.
- In *very long instruction word* (VLIW) CPU designs, parallelism is achieved by allowing multiple operations to be encoded into a single very wide instruction word, for the purpose of executing them in parallel.

Assembly Language

- a simple text-based version of machine language was developed called *assembly language*
- each instruction within a given CPU's ISA is given a *mnemonic*
- Registers can be referred to by name (e.g., R0 or EAX), and memory addresses can be written in hex, or assigned symbolic names
- A tool known as an *assembler* reads the program source file and converts it into the numeric ML representation understood by the CPU.

Machine Language

- ALU and MMU instructed to do useful things by a **program** encoded as an **instruction stream**
- Each instruction performs one simple(ish) operation
  - **Move** data between registers and memory
  - Perform **arithmetic** (add, sub, mul, div, …)
  - Perform **logical** ops (bit shift, and, or, ...)
    - Typically inputs and outputs are registers
  - **Branch** (i.e., change contents of instruction pointer, IP)
    - Conditional (based on **bits** in **status register**)
    - Unconditional (**jump**)

Machine Language

- Each ML instruction comprised of:
  - **Opcode**: ***which*** operation to perform
  - **Addressing mode flags**: ***how*** to perform the operation
  - **Operands**: on ***what data*** should CPU operate
- An ML instruction is encoded into an **instruction word**
  - Can be fixed-width or variable width

C	Opcode	AM	Operand 0	Operand 1	
C	Opcode	AM	Operand 0	Operand 1	
E	Opcode	AM	Operand 0	Operand 1	

Assembly Language

- Hard to remember numeric opcodes, modifier flags and how to properly encode operands into instruction words!
- **Assembly language** uses ***textual*** programming language to encode ML program
  - **Mnemonics** for opcodes
    - **mov** = move data between memory and registers
    - **mul** = multiply
    - **jmp** = jump; **bz** = branch if zero
  - Use register **names**
  - Can **label** branch targets and global variables

Example

```
if (a > b)
return a + b;
else
return 0;
```

```
; if (a > b)
cmp eax, ebx ; compare the values
jle ReturnZero ; jump if less than or equal
; return a + b;
add eax, ebx ; add & store result in EAX
ret ; (EAX is the return value)
ReturnZero:
; else return 0;
xor eax, eax ; set EAX to zero
ret ; (EAX is the return value)
```

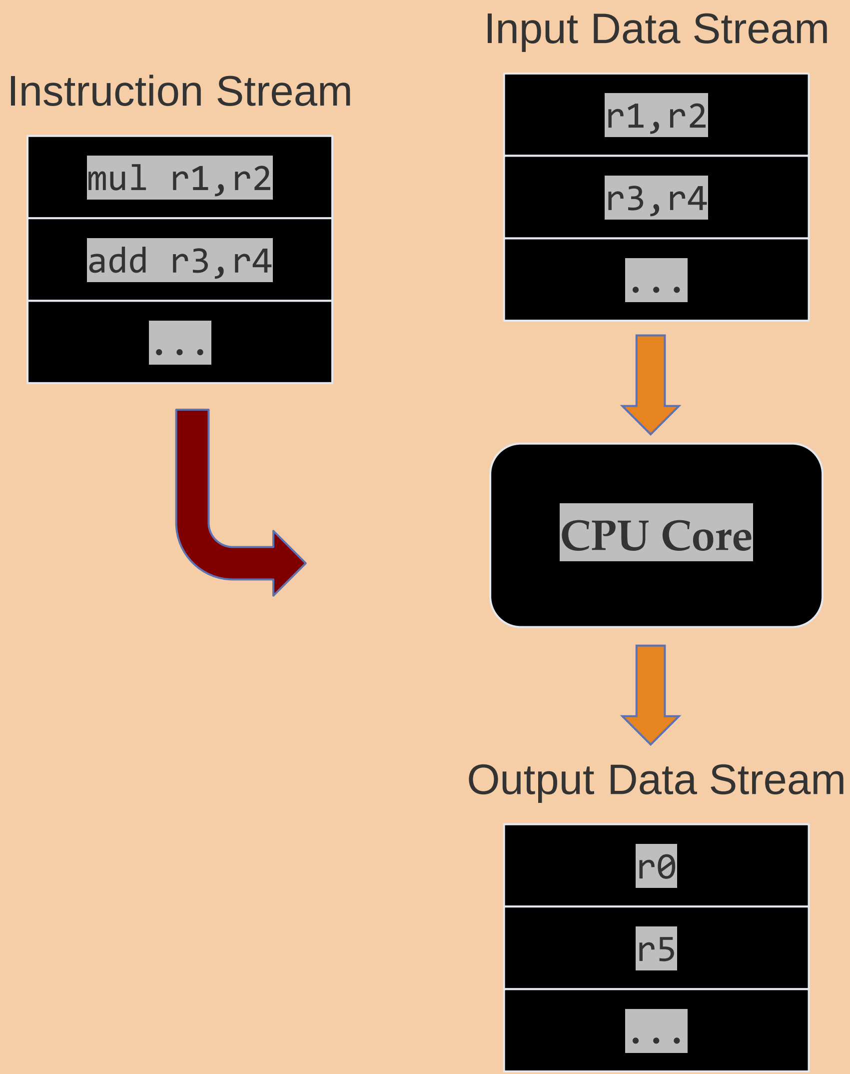
Assembly Language

```
AddIfGreater:
    ; function prologue
    push  ebp        ; push base ptr (EBP) onto stack
    mov   ebp, esp   ; set EBP to cur stack top (ESP)
    push  ebx        ; also save EBX because we use it

    ; load arguments a and b into registers EAX and EBX
    mov   eax, dword ptr [ebp+8]
    mov   ebx, dword ptr [ebp+12]

    ; if (a > b)
    cmp   eax, ebx   ; compare the values
    jle   ReturnZero ; jump if less than or equal

    ; return a + b;
    add   eax, ebx   ; add & store result in EAX
    jmp   Done       ; (EAX is the return value)

ReturnZero:
    ; else return 0;
    xor   eax, eax   ; set EAX to zero (return value)

Done:
    ; function epilogue: restore registers and return
    pop   ebx
    pop   ebp
    ret
```

Instruction Set Architecture (ISA)

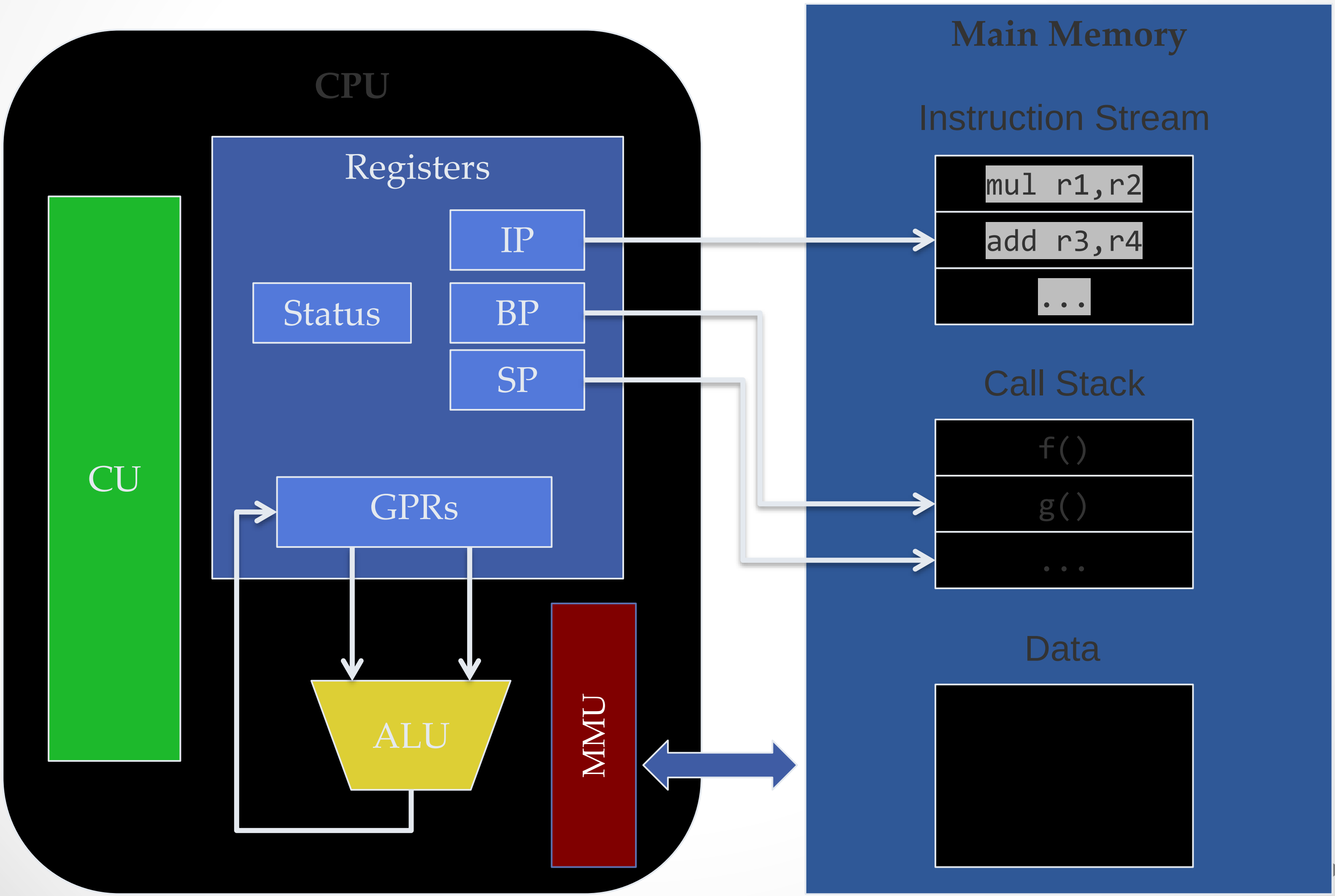
- Each CPU provides a different **instruction set architecture** (ISA)
- ISA defines:
  - Set of **opcodes** recognized by CPU
  - **Number** and **names** of registers
  - **Addressing modes** supported by CPU
  - Exposes *some* details about the **hardware**
    - Does the CPU contain an **FPU**? a **VPU**?
    - How is **I/O** done? Memory-mapped? Register-based?
    - Can it issue **multiple instructions** per clock? (VLIW)
    - **Privileged mode** supported? How many protection rings?

Execution Context



- **Thread** =
  - Running **instance** of an instruction stream
  - Single **flow of control**
- **Execution context** =
  - Instruction stream (IP)
  - Contents of all registers
  - Call stack
- **CPU core** =
  - Hardware components needed to execute a thread

Execution Context



Addressing Modes

- "move" (which transfers data between registers and memory) has many different variants.
  - Are we moving a value from one register to another? Are we loading a literal value like 5 into a register? Are we loading a value from memory into a register? Or are we storing a value in a register out to memory?
- *Register addressing* -> one register to another.
- *Immediate addressing* -> literal value to a register.
- *Direct addressing* -> from memory.
- *Register indirect addressing* -> the target memory address is taken from a register (pointers)
- *Relative addressing* -> the target memory address is specified as an operand, and the value stored in a specified register is used as an offset from that target memory address (indexed arrays).

Memory Architectures

- memory is a single homogeneous block (von Neumann computer architecture)
- Whenever a physical memory device is assigned to a range of addresses in a computer's address space, we say that the address range has been *mapped* to the memory device.
- A 64-bit address bus can access 16 EiB (ExbiByte = 1024^6) of memory
- an address range might also be mapped to other *peripheral devices*, such as a joypad or a network interace card (NIC).
- the CPU can perform I/O operations on a peripheral device by reading or writing to addresses
- a CPU might communicate with non-memory devices via special registers known as *ports*

The Apple II Memory Map

- The Apple II had a 16-bit address bus, meaning that its address space was only 64 KiB in size.
- This address space was mapped to ROM, RAM, memory-mapped I/O devices and video RAM regions as follows:
  - 0xC100 - 0xFFFF ROM (Firmware)
  - 0xC000 - 0xC0FF Memory-Mapped I/O
  - 0x6000 - 0xBFFF General-purpose RAM
  - 0x4000 - 0x5FFF High-res video RAM (page 2)
  - 0x2000 - 0x3FFF High-res video RAM (page 1)
  - 0x0C00 - 0x1FFF General-purpose RAM
  - 0x0800 - 0x0BFF Text/lo-res video RAM (page 2)
  - 0x0400 - 0x07FF Text/lo-res video RAM (page 1)
  - 0x0200 - 0x03FF General-purpose and reserved RAM
  - 0x0100 - 0x01FF Program stack
  - 0x0000 - 0x00FF Zero page (mostly reserved for DOS)

Virtual Memory

- In today's operating systems, programs work in terms of *virtual* addresses rather than *physical* addresses.
- It allows programs to make use of more memory than is actually installed in the computer, because data can overflow from physical RAM onto disk.
- In a *virtual memory system,* address is first *remapped* by the CPU via a look-up table that's maintained by the OS.
- The remapped address might end up referring to an actual cell in memory (with a totally different numerical address).
- In a virtual memory system, the addresses used by programs are called *virtual addresses*

Video RAM

- A range of memory addresses assigned for use by a video controller is known as *video RAM* (VRAM).
- PlayStation 4 and the Xbox One, both the CPU and GPU share access to a single, large block of unified memory.
- In personal computers, the GPU often lives on a separate circuit board
- In personal computers, a bus protocol such as PCI, AGP or PCI Express (PCIe) is used to transfer data back and forth between "main RAM" and VRAM via the expansion slot's bus.

Virtual Memory Pages

- The entire addressable memory space (that's 2^n byte-sized cells if the address bus is *n* bits wide) is conceptually divided into equally-sized contiguous chunks known as *pages*.
- Page sizes differ from OS to OS, but are always a power of two—a typical page size is 4 KiB or 8 KiB
- Assuming a 4 KiB page size, a 32-bit address space would be divided up into 1,048,576 distinct pages, numbered from 0x0 to 0xFFFFF

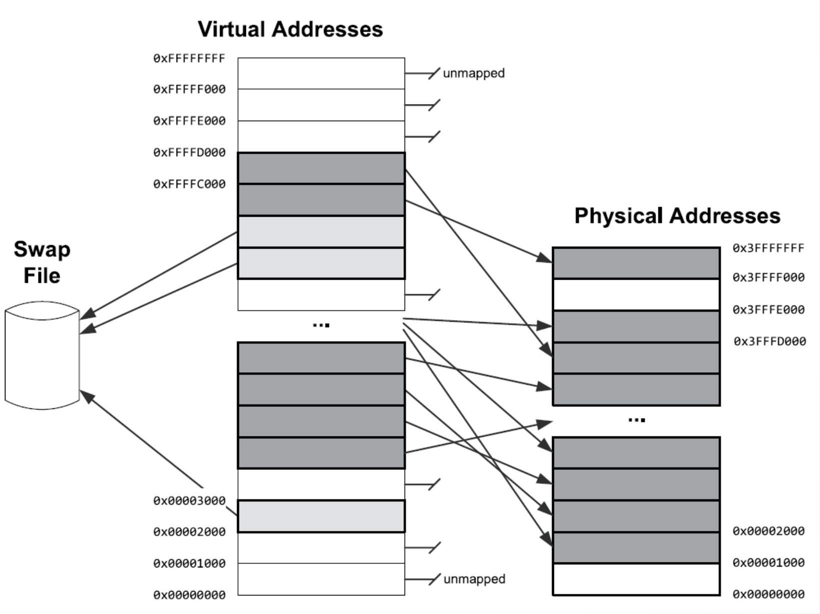
From Address	To Address	Page Index
0x00000000	0x00000FFF	Page 0x0
0x00001000	0x00001FFF	Page 0x1
0x00002000	0x00002FFF	Page 0x2
	...	
0x7FFF2000	0x7FFF2FFF	Page 0x7FFF2
0x7FFF3000	0x7FFF3FFF	Page 0x7FFF3
	...	
0xFFFFE000	0xFFFFEFFF	Page 0xFFFFE
0xFFFFF000	0xFFFFFFFF	Page 0xFFFFF

Virtual to Physical Address Translation

- the address is split into two parts: the *page index* and an *offset* within that page
- The page index is then looked up by the CPU's *memory management unit* (MMU) in a *page table* that maps virtual page indices to physical ones.
- For a page size of 4 KiB, the offset is just the lower 12 bits of the address, and the page index is the upper 20 bits, masked off and shifted to the right by 12 bits.
- the virtual address 0x1A7C6310 corresponds to an offset of 0x310 and a page index of 0x1A7C6
- if virtual page 0x1A7C6 happens to map to physical page 0x73BB9, then the translated physical address would end up being 0x73BB9310.

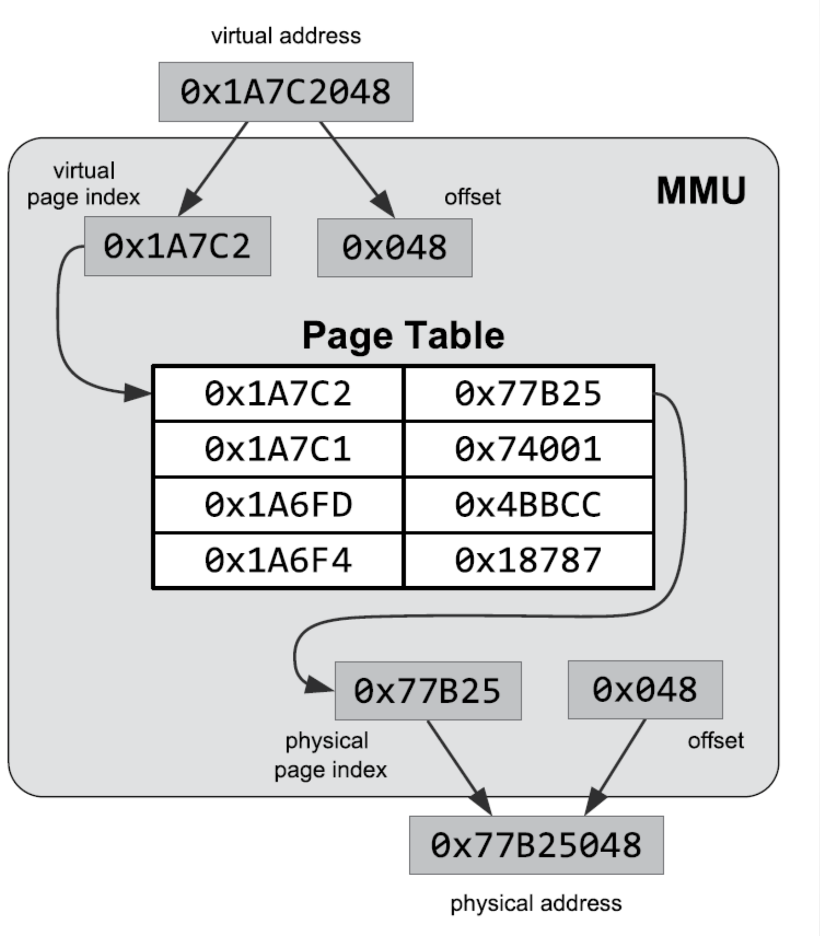
Page Fault

- Page-sized address ranges in a virtual memory space are remapped either to physical memory pages, a swap file on disk, or they may remain unmapped.
- If the page table indicates that a page is not mapped to physical RAM, the MMU raises an *interrupt*, which tells the operating system that the memory request can't be fulfilled. This is called a *page fault*.



memory management unit (MMU)

- The MMU intercepts a memory read operation, and breaks the virtual address into a virtual page index and an offset.
- The virtual page index is converted to a physical page index via the page table, and the physical address is constructed from the physical page index and the original offset.
- Finally, the instruction is executed using the remapped physical address.

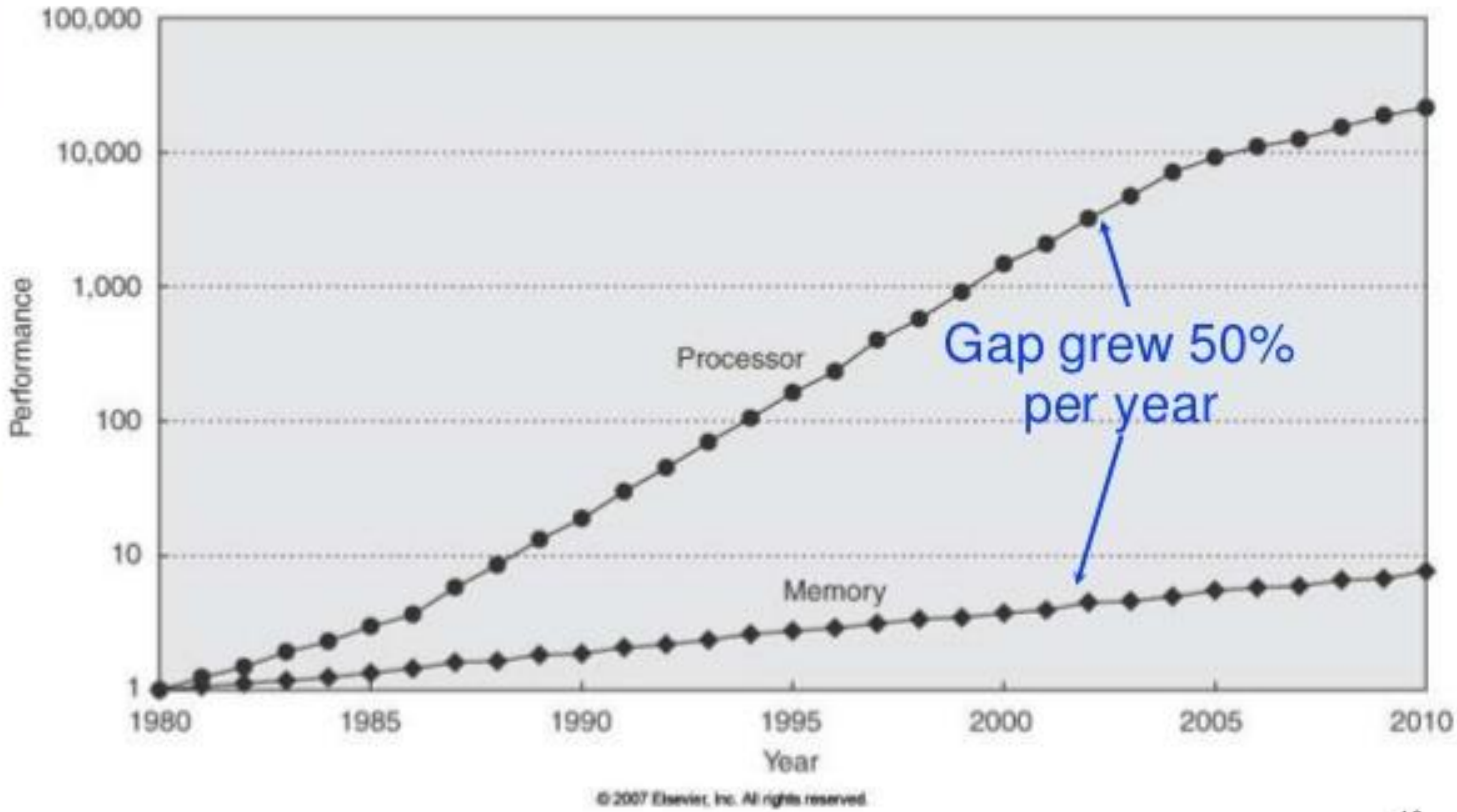


The Translation Lookaside Buffer (TLB)

- page sizes are small relative to the total size of addressable memory (typically 4 KiB or 8 KiB)
- the page table can become very large
- Looking up physical addresses in the page table would be time-consuming
- A small table known as the *translation lookaside buffer* (TLB) is maintained within the MMU
- To speed up access, a caching mechanism is used, based on the assumption that an average program will tend to reuse addresses within a relatively small number of pages, rather than read and write randomly across the entire address range.

Memory Architectures

Processor Memory Gap



Source: Computer Architecture, A Quantitative Approach by John L. Hennessy and David A. Patterson

Memory Cache Hierarchies

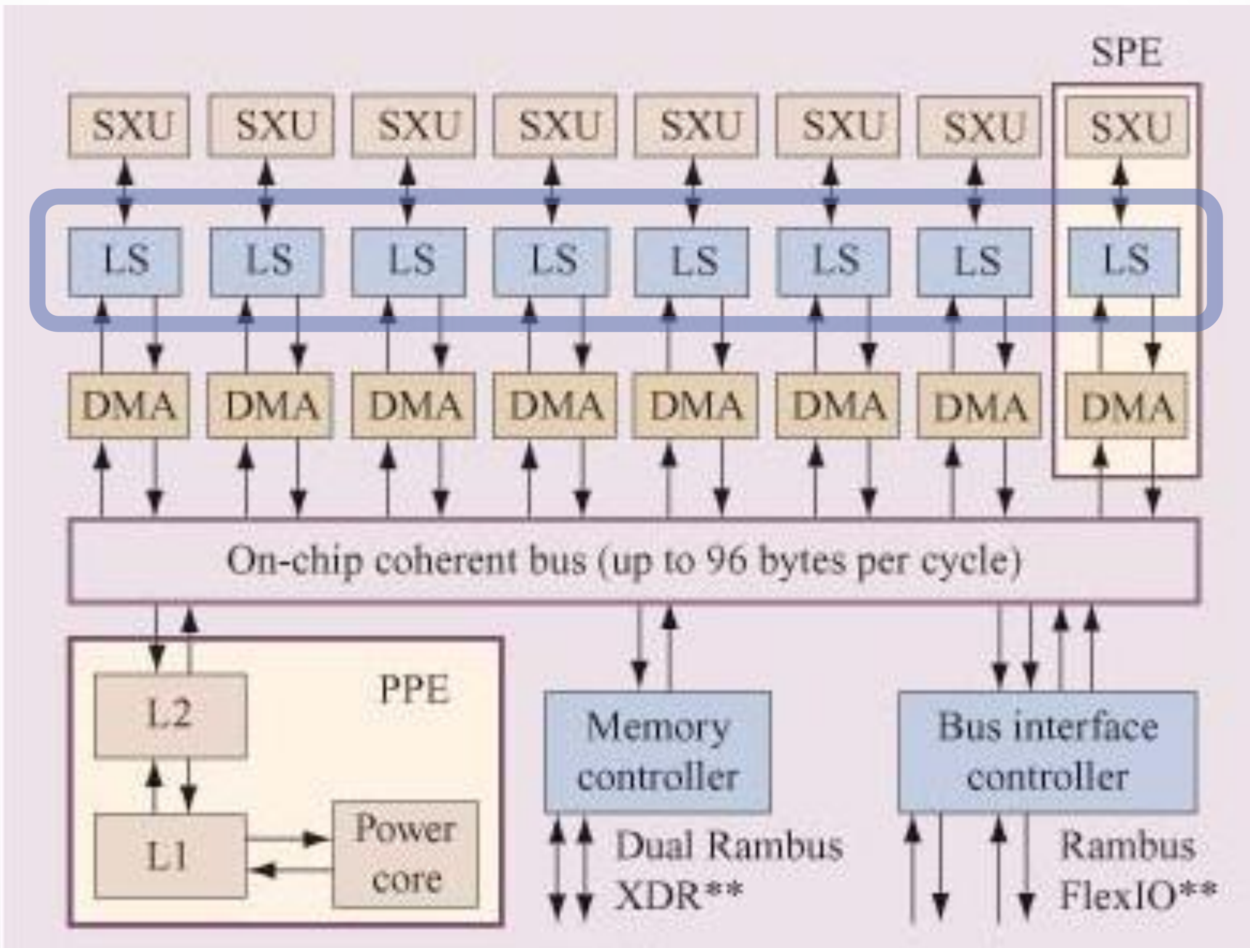
- **Memory access latency** = the time between:
  - requesting data from memory controller and
  - that data arriving in a CPU register
- Latency highly dependent on **proximity** to CPU core
  - Register latency: 1 cycle
  - Main RAM latency: 200+ cycles

Nonuniform Memory Access (NUMA)

- One way to reduce memory latency is to give each CPU its own local memory bank
  - e.g., **local stores** for each SPU on **PS3**

Nonuniform Memory Access (NUMA)

- PS3 cell architecture



Nonuniform Memory Access (NUMA)

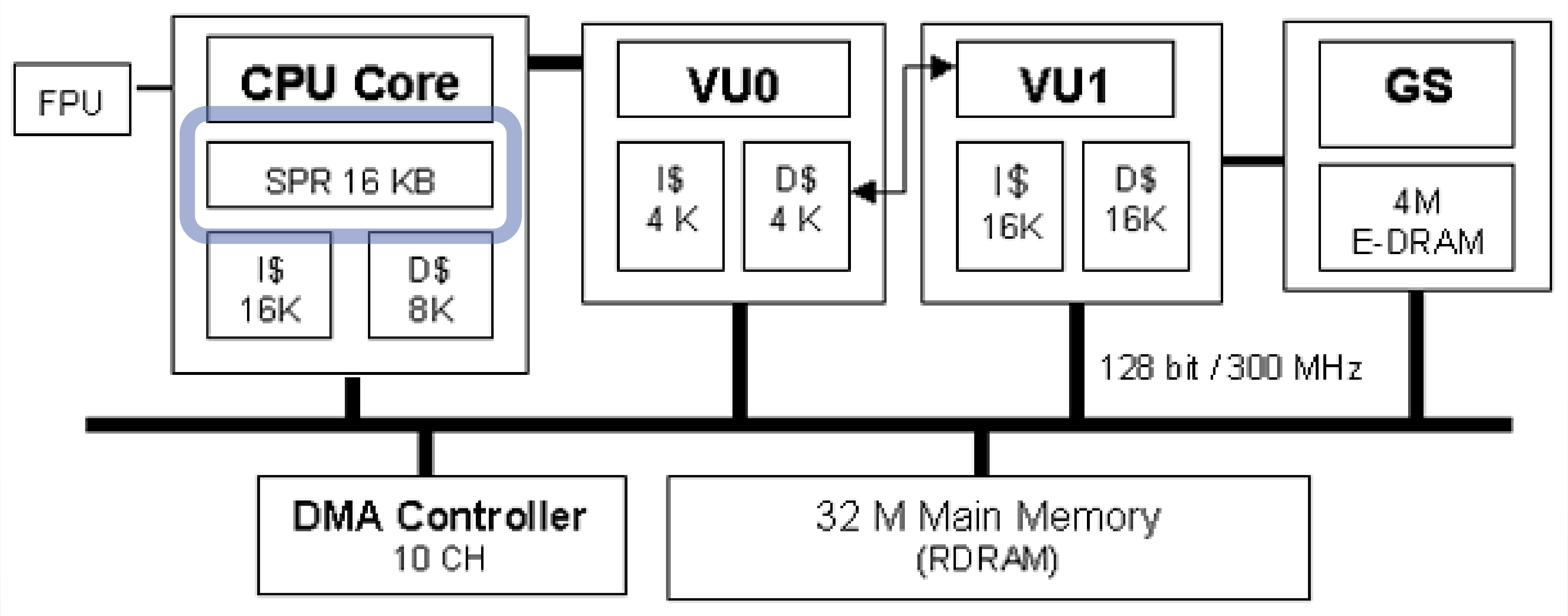
- PS3's **DMA controller** (DMAC) is like a little coprocessor whose only job is to ferry data around on the system buses
  - A form of **parallelism**!

Nonuniform Memory Access (NUMA)

- Another example is the **scratchpad** on PS2
  - Scratchpad memory wasn't actually *faster* to access by CPU
  - But it could be accessed by CPU **directly**, without using system address and data **buses**
  - As a result, CPU could be **busy doing work** with data in scratchpad while the buses and DMAC were busy transferring data

Nonuniform Memory Access (NUMA)

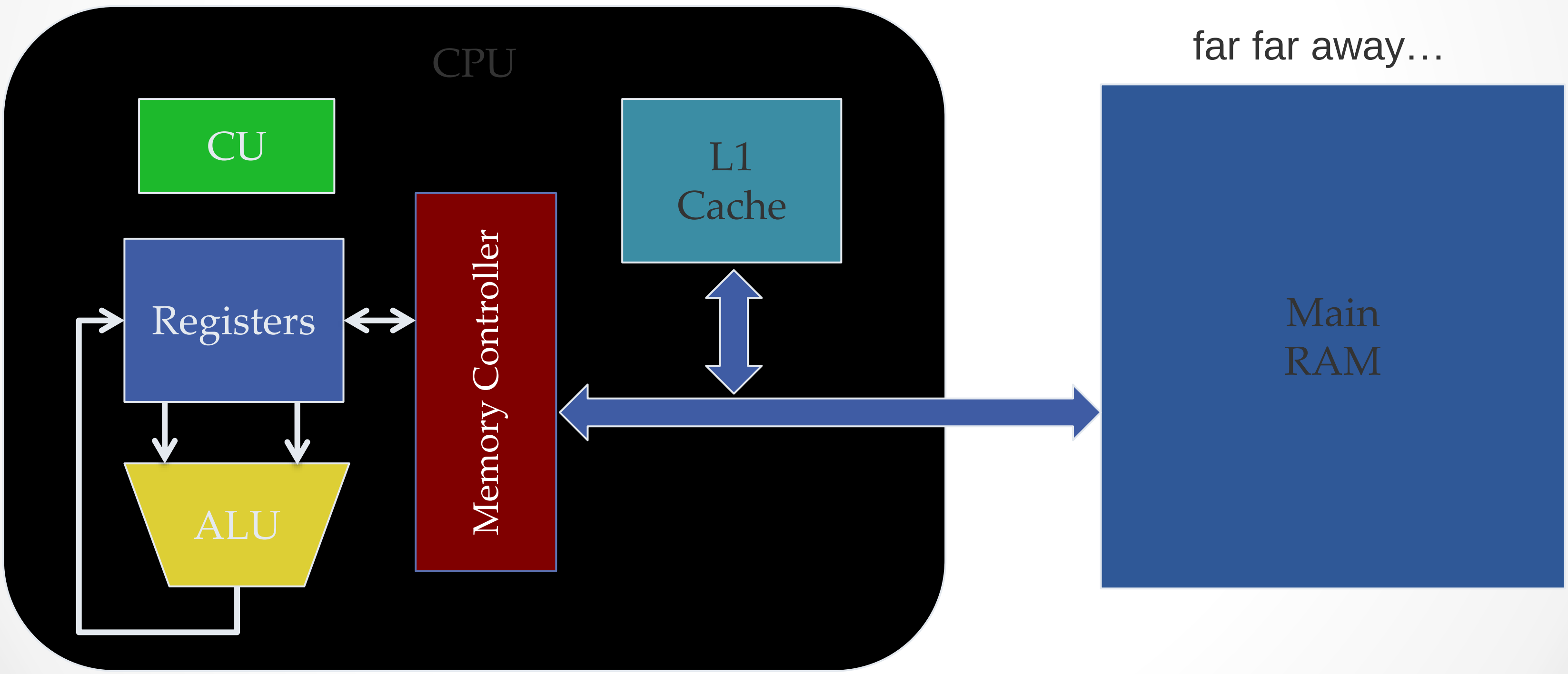
- PS2 architecture, showing scratchpad (SPR)



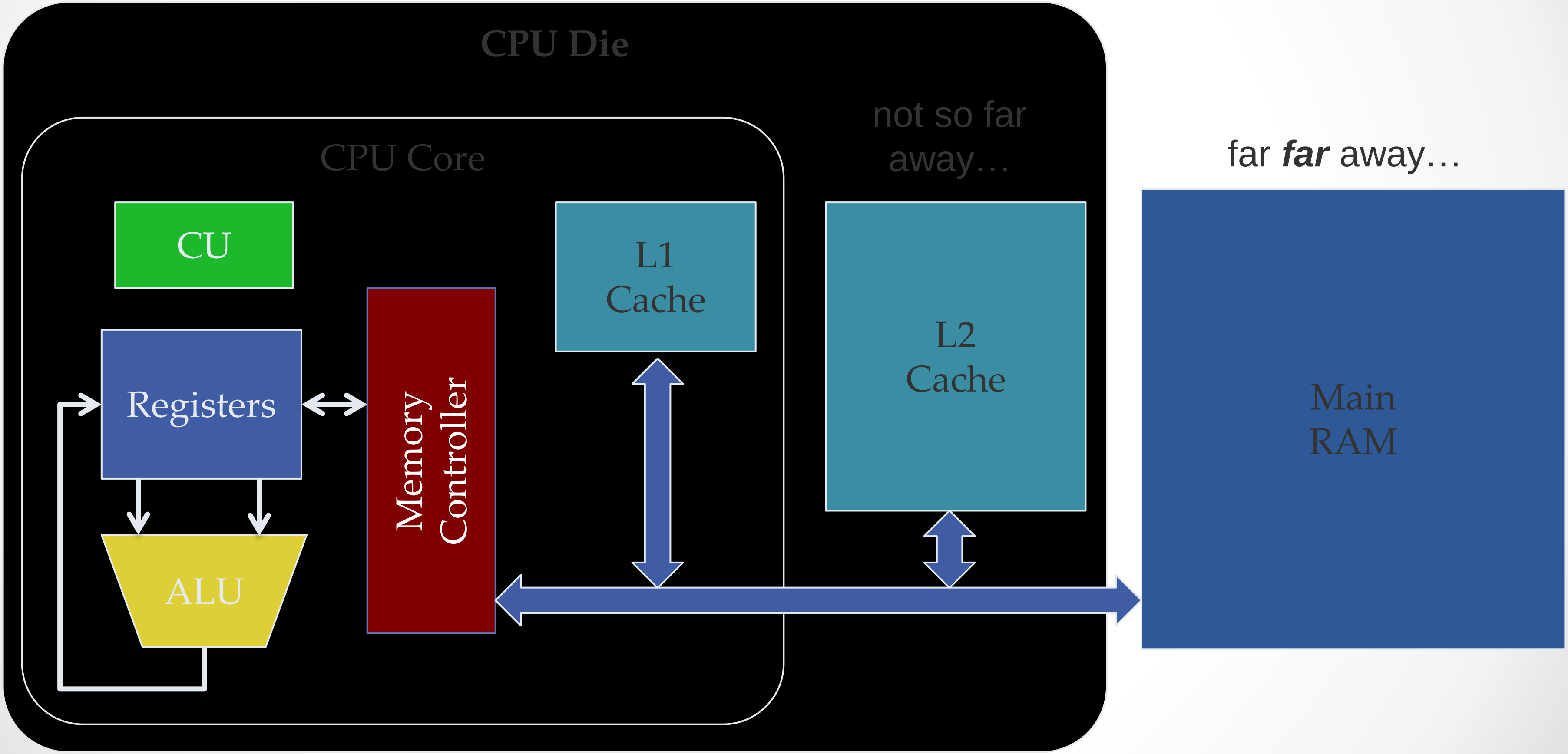
Memory Cache Hierarchies

- Local stores require explicit DMA, hard to program
- Can we use this same idea, but make it "automatic"?
- Keep **most-recently used data** in a **cache** that is:
    - Closer to CPU core (for lower latency, like local stores)
    - Smaller than main RAM (to keep costs in control)

Memory Cache Hierarchies



Memory Cache Hierarchies

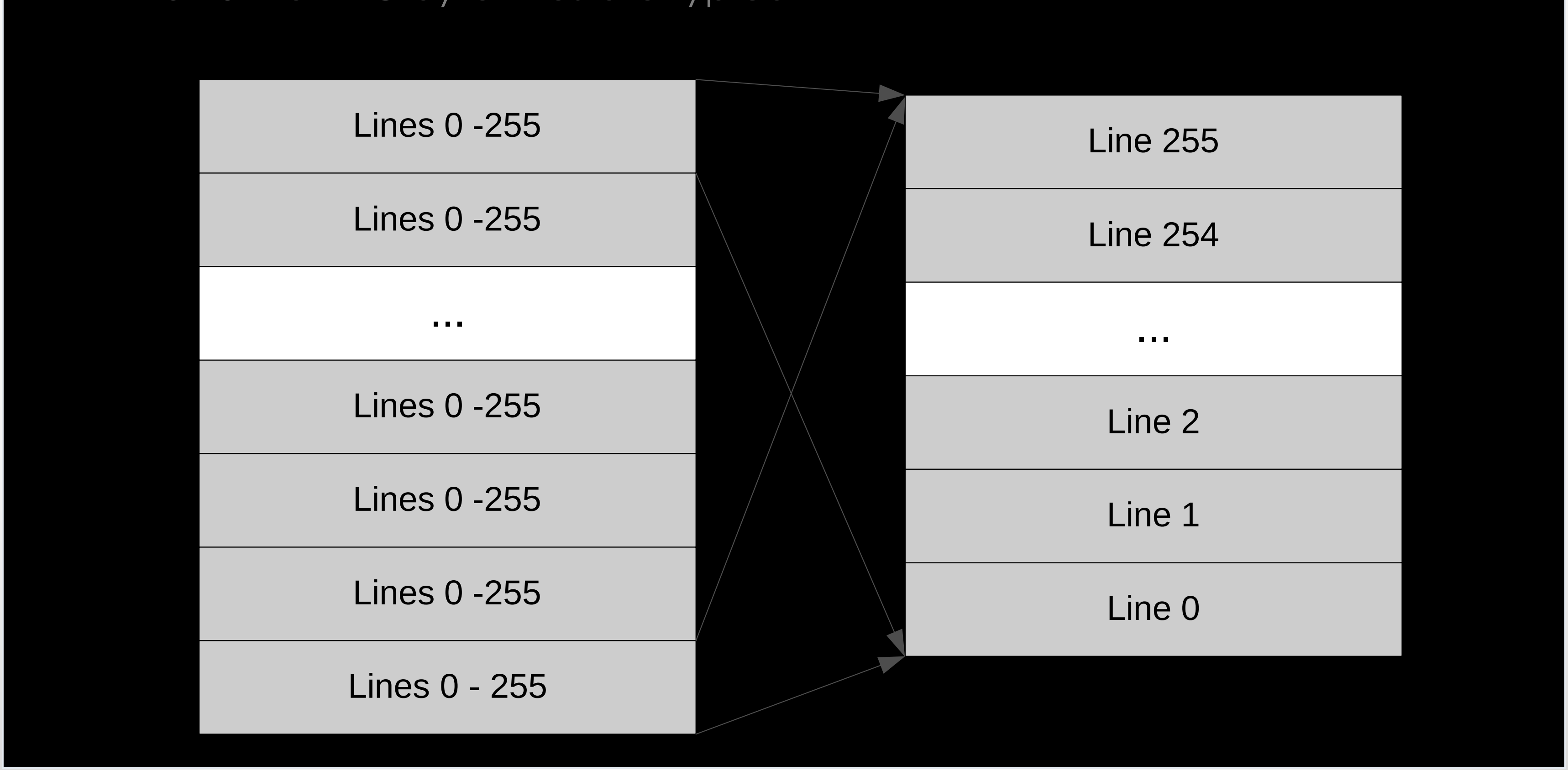


Memory Cache Hierarchies

- Memory cache hierarchies reduce **average** memory access latency…
  - by taking advantage of **temporal** and **spatial locality**
- **Temporal locality**
  - Data tends to be accessed repeatedly in a short time window
  - If a program accesses address x, there's a good chance it'll access address x again in the near future
- **Spatial locality**
  - Data tends to be accessed sequentially, or in blocks
  - If program accesses address x, there's a good chance it'll access address $x + n$ as well (for small $|n|$)

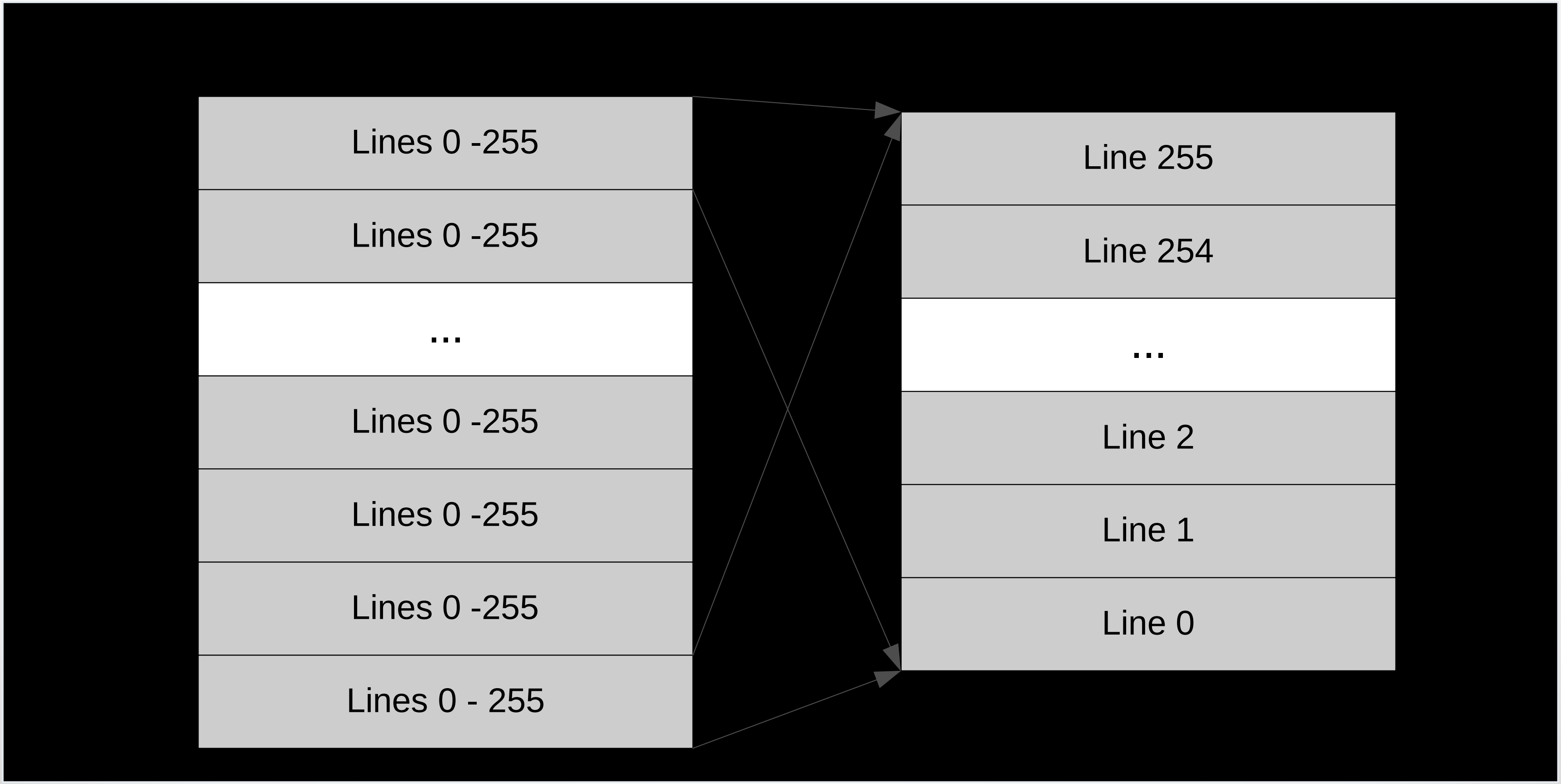
Memory Cache Hierarchies

- Caches work by dividing main memory into **lines**
  - 64- or 128-byte lines are typical



Memory Cache Hierarchies

- **Lines** of main RAM can be read into the **cache**
  - Once in the cache, CPU can access the data **more quickly**



Memory Cache Hierarchies

- Once a line is in the cache, how do we know from which memory line it came?
  - **Line index**
    - 64-byte cache: **`(addr & 0x3F)`**
    - 128-byte cache: **`(addr & 0x7F)`**
  - **Tag**
    - 64-byte cache: **`(addr >> 6)`**
    - 128-byte cache: **`(addr >> 7)`**
  - Store the tag with each cache line to keep track of its original location in memory

Memory Cache Hierarchies

- **Tags** are stored with each line in cache to keep track of from whence each line came

Memory Cache Hierarchies

- When CPU **reads** a data item (single byte or larger):
  - Address converted into **line index**
- Memory controller checks L1 cache: Does cache **already contain** this line?
  - **Hit:** If line is present, fetch data from the line
  - **Miss:** If not present, fetch from **next level** cache (L2)…
    - Rinse and repeat until main memory is reached
- On a multicore system, read requests may also be fulfilled by **other cores** at the *same* level

Memory Cache Hierarchies

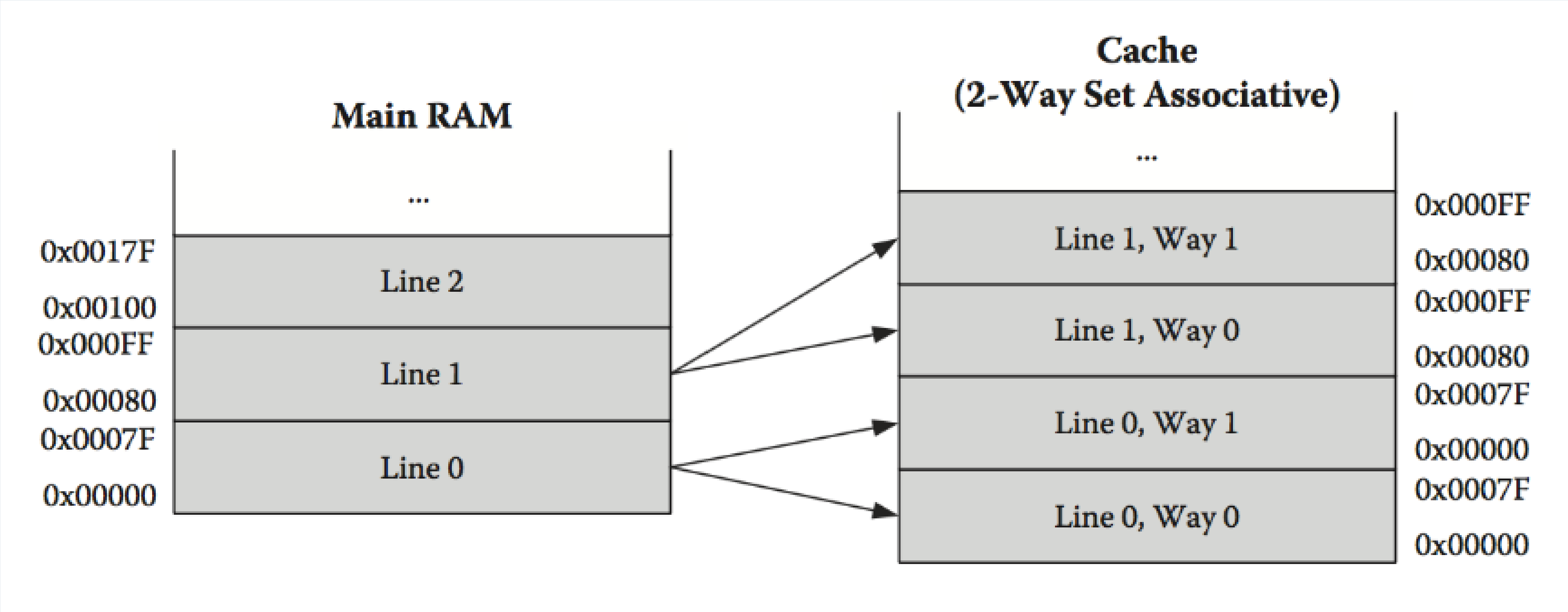
- When CPU **writes** a data item (single byte or larger):
  - Address converted into line index
- Memory controller checks L1 cache: Does cache already contain this line?
  - **Hit:** If line is present, **write** item into line, mark line **modified**
    - (This gets more complicated in a *multicore* machine…)
  - **Miss:** If not present, **fetch** line from L2, L3, … main memory, write and mark modified
- Writes to cache lines not necessarily **written back** to main RAM immediately
  - **Write-back** triggered on next **read** or **invalidate** of cache line
- **Write-through** operation can bypass cache

Memory Cache Hierarchies

- What we've just described is a **direct-mapped** cache
  - Each line in memory maps to exactly one line in cache
  - Conflicts are likely (e.g., addresses 0x80, 0x100 and 0x180)
- **Fully associative** cache
  - Any line in memory can be placed *anywhere* in the cache
  - Requires **linear search** of tags to find lines in the cache
- ***n*-way set associative** cache
  - Each line in memory maps to *n* lines in cache
  - Best of both worlds:
    - Reduces line conflicts by factor of *n*
    - Limited searching (only search the ways, not entire cache)

Memory Cache Hierarchies

- e.g., **2-way set associative** cache
  - Each line in memory maps to 2 lines in cache



Memory Cache Hierarchies

- What happens when cache becomes **full**?
  - Must **evict** previous data to make room
- Cache line **replacement policy**
  - FIFO (first in, first out)
    - Only option in a direct-mapped cache
  - NMRU (not most-recently used): 1 bit per set
  - LRU (least-recently used): costly for $n > 2$
  - LFU (least-frequently used)
  - Pseudo-random (!)
  - Optimal?
- https://ece752.ece.wisc.edu/lect11-cache-replacement.pdf

Memory Cache Hierarchies

- Memory access latencies on PS4
  - **Registers**: 1 cycle
  - **L1 cache**: 4 cycles
  - **L2 cache**: 26 cycles (190 cycles ***between*** 4-core clusters)
  - **Main RAM**: 200+ cycles
- L1 cache usually split into two distinct caches:
  - **Instruction cache** (I$)
  - **Data cache** (D$)
    - This prevents code from degrading data cache performance, and vice-versa
    - e.g., a big loop iterating over array of small data items

Memory Cache Hierarchies

- So why do we care about caches?
  - Aren't they "automagic"?
  - Can't programmers just ignore them (trust that they work) and get on with programming?

Discussion: 5 mins

Memory Cache Hierarchies

So why do we care about caches?

- Understanding caches is an **optimization tool**
  - Structure **data** to avoid excessive cache misses
    - Pack data into non-sparse arrays
    - Avoid skipping around in memory

Memory Cache Hierarchies

- Consider a particle system in which each particle is defined like this:

```
class Particle
{
    ParticleId    m_globalId;
    Point         m_pos;
    Quat          m_rot;
    Vector        m_scale;
    Color32       m_color;
    Texture*      m_apTex[4];
    Vec2          m_aUV[4];
    AnimConfig    m_animConfig;
    // ...
};
```

`sizeof``(Point) ==` **`16`**

Therefore, only 4 **`Points`** per cache line, on a CPU with 64-byte lines.

Memory Cache Hierarchies

```
void ApplyWind(const Vector& windVel, Particle* aParticle,
               const int count)
{
    const float dt = GetFrameDeltaTime();
    const Vector windVelPerFrame = windVel * dt;

    for (int i = 0; i < count; ++i)
    {
        Particle& part = aParticle[i];
        part.m_pos += windVelPerFrame; // cache miss every time!
    }
}
```

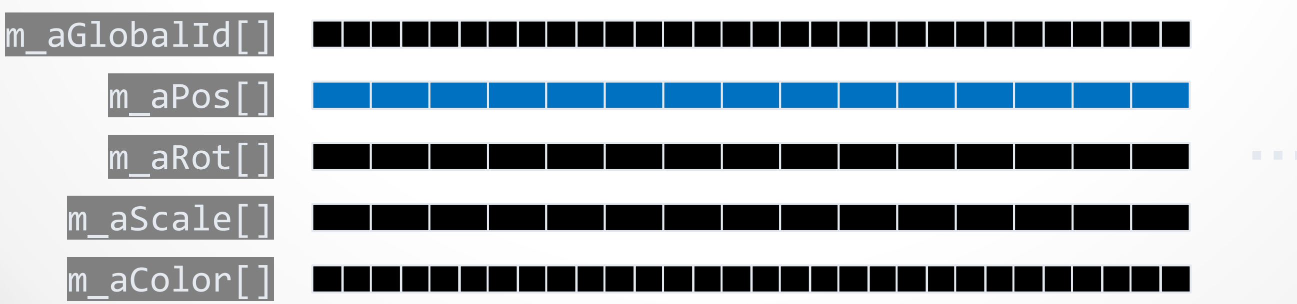
Memory Cache Hierarchies

- Let's rearrange the data to improve cache performance

```
class ParticleGroup
{
    I32            m_count;
    ParticleId*    m_aGlobalId;
    Point*         m_aPos;
    Quat*          m_aRot;
    Vector*        m_aScale;
    Color32*       m_aColor;
    Texture**      m_apTex; // 4 per particle
    Vec2*          m_aUV;   // 4 per particle
    U16*           m_aAnimConfigIndex;
    // ...
};
```

Memory Cache Hierarchies

```
void ApplyWind(const Vector& windVel,
               ParticleGroup& group)
{
    const float dt = GetFrameDeltaTime();
    const Vector windVelPerFrame = windVel * dt;

    for (int i = 0; i < group.m_count; ++i)
    {
        group.m_aPos[i] += windVelPerFrame;
    }
}
```



Memory Cache Hierarchies

So why do we care about caches?

- Understanding caches is an **optimization tool**
  - Structure **code** to avoid excessive cache misses **too**
    - Keep time-critical loops small in terms of code size
    - Inlining can be helpful, but can also lead to code bloat
    - Avoid virtual function calls in tight loops

Memory Cache Hierarchies

```
void UpdateParticles(Particle* aParticle, const int count)
{
    const float dt = GetFrameDeltaTime();

    for (int i = 0; i < count; ++i)
    {
        Particle& part = aParticle[i];
        AnimatePosition(part.m_pos, part.m_animConfig, dt);
        AnimateRotation(part.m_rot , part.m_animConfig, dt);
        AnimateColor(part.m_color , part.m_animConfig, dt);
        ApplyGravity(part.m_pos, dt);
        ApplyWind(part.m_pos, dt);
        // ...
    }
    // If the body of this loop doesn't fit in I$, we
    // could be getting I$ misses on every iteration.
}
```

Memory Cache Hierarchies

```
void UpdateParticles(ParticleGroup& group)
{
    const float dt = GetFrameDeltaTime();

    // This approach could be much more I$ friendly...

    for (int i = 0; i < group.count; ++i)
    {
        const int iAnimConfig = part.m_aAnimConfigIndex[i];
        AnimatePosition(part.m_aPos[i], iAnimConfig, dt);
    }
    for (int i = 0; i < group.count; ++i)
    {
        const int iAnimConfig = part.m_aAnimConfigIndex[i];
        AnimateRotation(part.m_aRot[i], iAnimConfig, dt);
    }
    // ...
}
```

What about D$ issues with anim config indices?

Memory Cache Hierarchies

So why do we care about caches? (continued)

- Understanding caches is crucial for **concurrency**
  - Data intended to be “local” to a core/thread should be on its own cache line
  - Atomic instructions like compare-and-swap (CAS) operate on cache lines
  - Multiple cores share cache lines via MESI protocol
    - Memory barriers/fences
- **More on this topic later…**

Processes, Threads and the Kernel

The Kernel

- The **kernel** is the core of the OS
- Provides all of the low-level features of the OS
  - **Processes** and **threads**
  - **File** and **network I/O**
  - **Virtual memory** mapping and swap file management
  - **Scheduling** threads to run on available CPU cores (**preemptive** multithreading)
  - Handles most **interrupts**
  - Interfaces to hardware **drivers**
- Higher-level OS functionality provided by **services**
  - Themselves processes, many of which run in user mode

The Kernel

- The kernel…
  - Runs in **ring 0** (**privileged** mode) on the CPU
  - Works directly in terms of **physical memory addresses**
  - Large block of addresses are **reserved** for use by the kernel, called **kernel space**
    - e.g., under 32-bit Windows and Linux: upper **1** or **2 GiB**
  - Handles **interrupts** via interrupt service routines (ISRs)
    - Hardware and software interrupts
  - User-space programs request privileged services by making **system calls** (aka **kernel calls**)
    - Set up arguments, then trigger a **software interrupt**
    - Causes **context switch** into kernel—**expensive**! (1000 cycles)

Processes

- A process is a **running instance** of an **executable file** (.exe on Windows, .elf under Linux/MacOS)
- Executable file contains:
    - **Text** segment: Relocatable machine code
    - **Data** segment: Initialized global and static variables
    - **BSS** segment: Uninitialized globals and statics

Processes

- A process consists of:
  - Process id (PID)
  - Permissions (which user/group owns it, etc.)
  - Reference to **parent** process
  - Environment variables
  - Open file descriptors
  - Current working directory
  - Resources of managing inter-process synchronization and communication (pipes, semaphores, etc.)

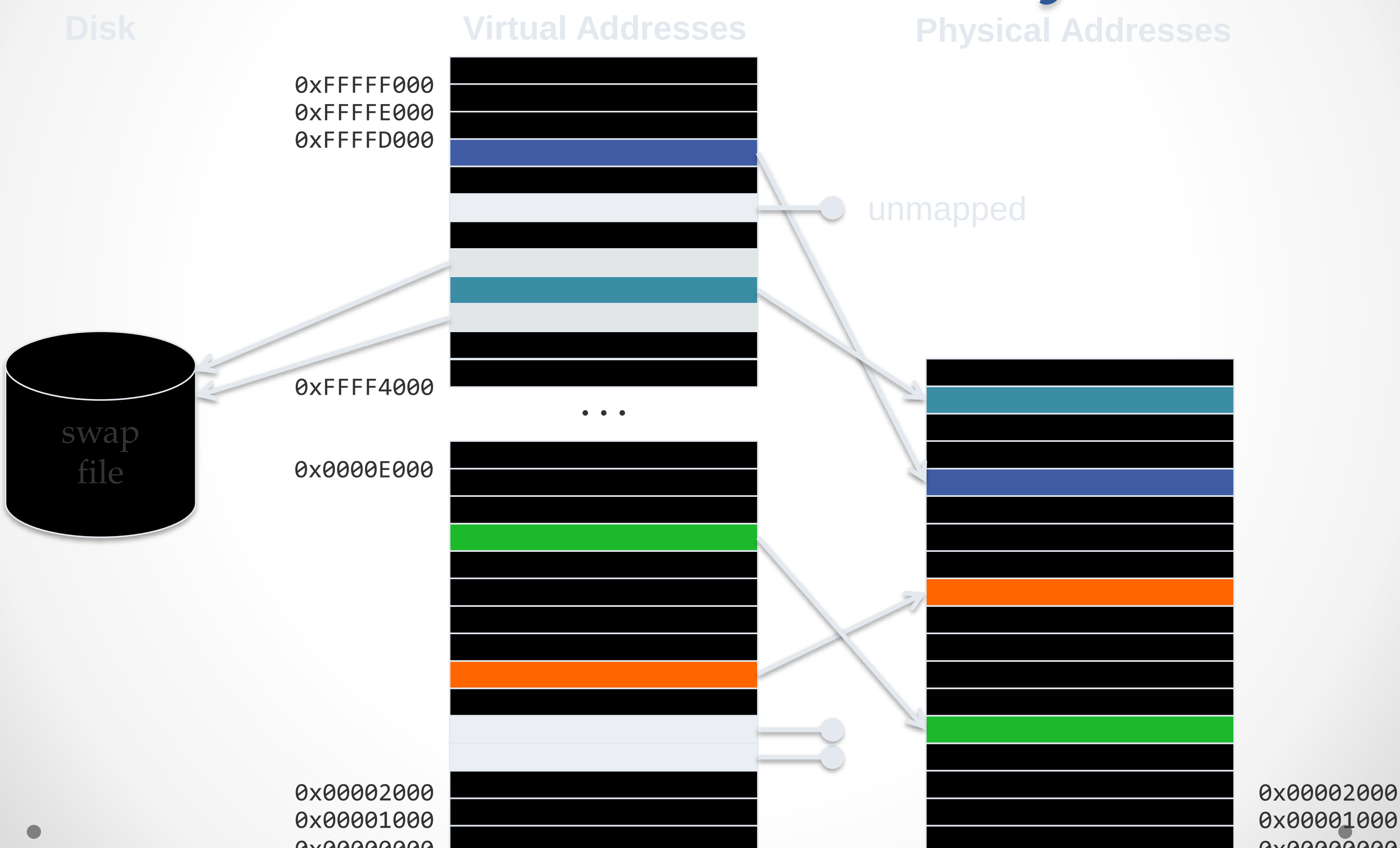
Processes

- A process consists of:
  - A **virtual memory space** containing:
    - Executable file **image** (text, data, BSS)
    - One or more **execution contexts** (**threads**), i.e. **call stacks**
    - **Heap** for dynamic allocation
  - One **thread** by default, but can spawn multiple threads

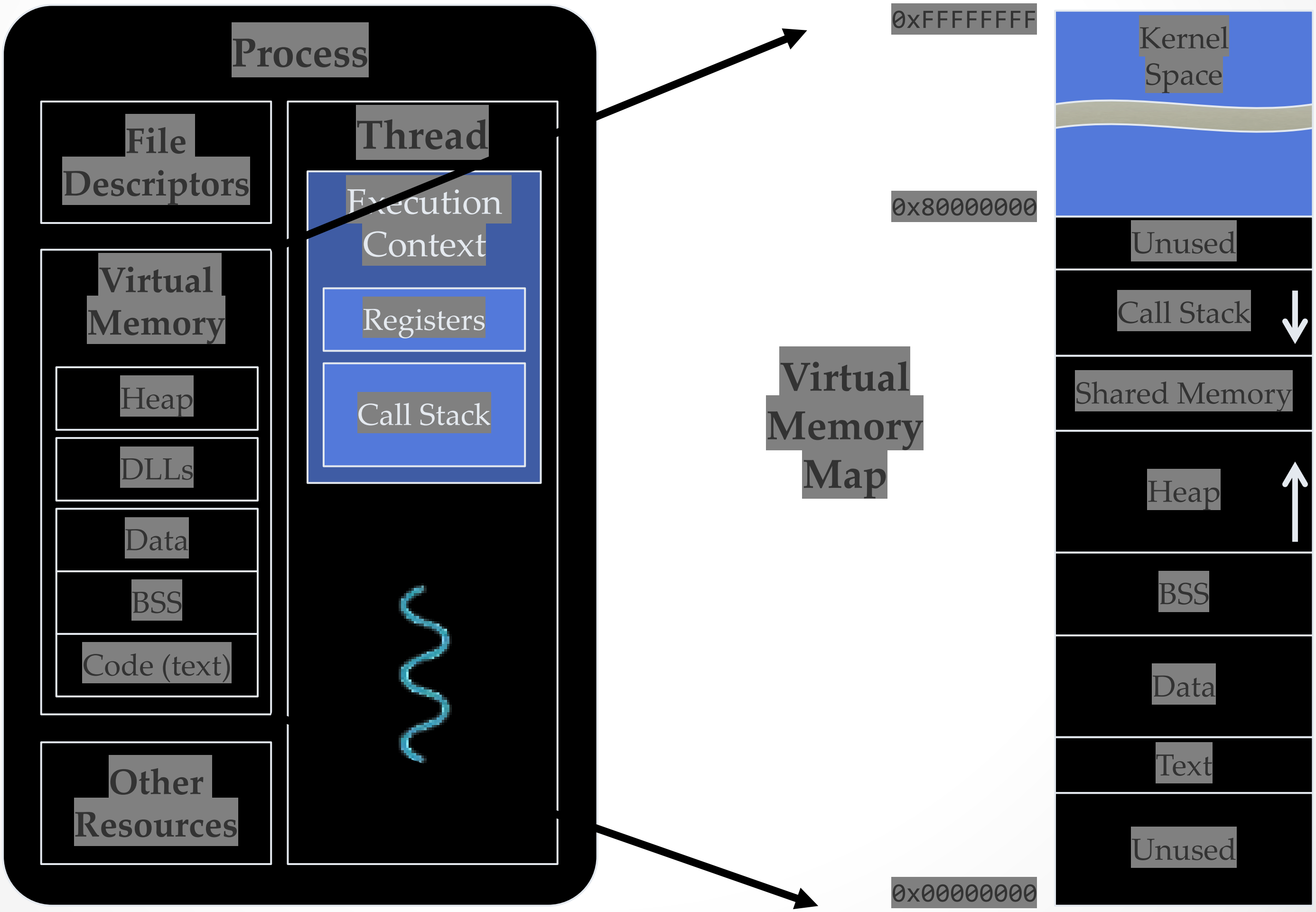
Virtual Memory

- Each process has its own private "view" of memory
  - User-space programs perform all memory accesses in terms of **virtual addresses**
  - The CPU and kernel cooperate in order to **map** virtual addresses to **physical addresses** at runtime

Virtual Memory



Processes

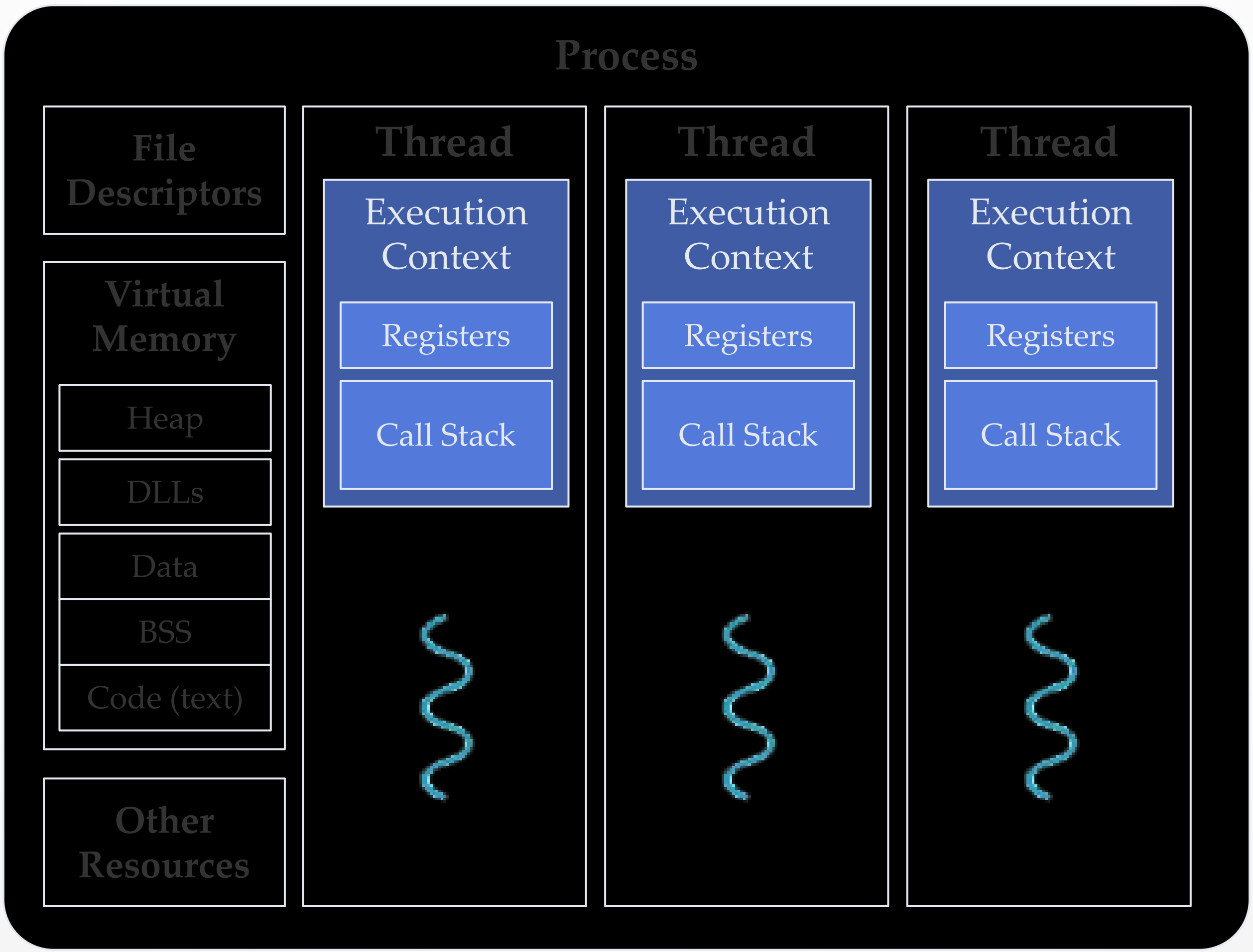


Process
File Descriptors
Virtual Memory
Heap
DLLs
Data
BSS
Code (text)
Other Resources
Thread
Execution Context
Registers
Call Stack
Virtual Memory Map
0xFFFFFFFF
0x80000000
0x00000000
Kernel Space
Unused
Call Stack
Shared Memory
Heap
BSS
Data
Text
Unused

Threads

- The kernel doesn't actually run processes
  - It runs **threads**!
  - A process contains one thread by default, but can spawn more
- A **thread** represents a **single instruction stream**
- A thread's **execution context** consists of:
  - A **call stack** (region of memory for stack frames, grows down)
  - **Registers**
    - IP (current instruction), SP and BP (stack frame), GPRs and status register
- All threads in a process **share** the same **virtual address space**

Threads and Processes



Thread API

- Each OS/kernel provides its own API for creating threads
  - IEEE POSIX 1003.1c threads (**pthreads**) is a standard API available on all UNIX favors and also Windows
  - **Windows** has a native API too, of course
  - **PS4 SDK**'s thread API is modeled after pthreads
  - **C++11** added threading to the **standard library**

Thread API (POSIX)

```
#include <pthread.h>

void entry_point(int i) { ... }

int main()
{
    pthread_t t1, t2;
    pthread_create(&t1, nullptr, entry_point, 0);
    pthread_create(&t2, nullptr, entry_point, 1);

    // do some other useful work while threads run...

    pthread_join(&t1);
    pthread_join(&t1);
    // at this point, both threads have terminated
}
```

Thread API (C++11)

```
#include <thread>

void entry_point(int i) { ... }

int main()
{
    std::thread t1(entry_point, 0);
    std::thread t2(entry_point, 1);

    // do some other useful work while threads run...

    t1.join();
    t2.join();
    // at this point, both threads have terminated
}
```

Thread API (Windows)

```
#include <windows.h>

DWORD WINAPI entry_point(LPVOID lpParam) { ... }

int main()
{
    HANDLE ahT[2];
    ahT[0] = CreateThread(NULL, 0, entry_point, …);
    ahT[1] = CreateThread(NULL, 0, entry_point, …);

    // do some other useful work while threads run...

    WaitForMultipleObjects(2, ahT);
    // at this point, both threads have terminated
}
```

Thread Scheduling

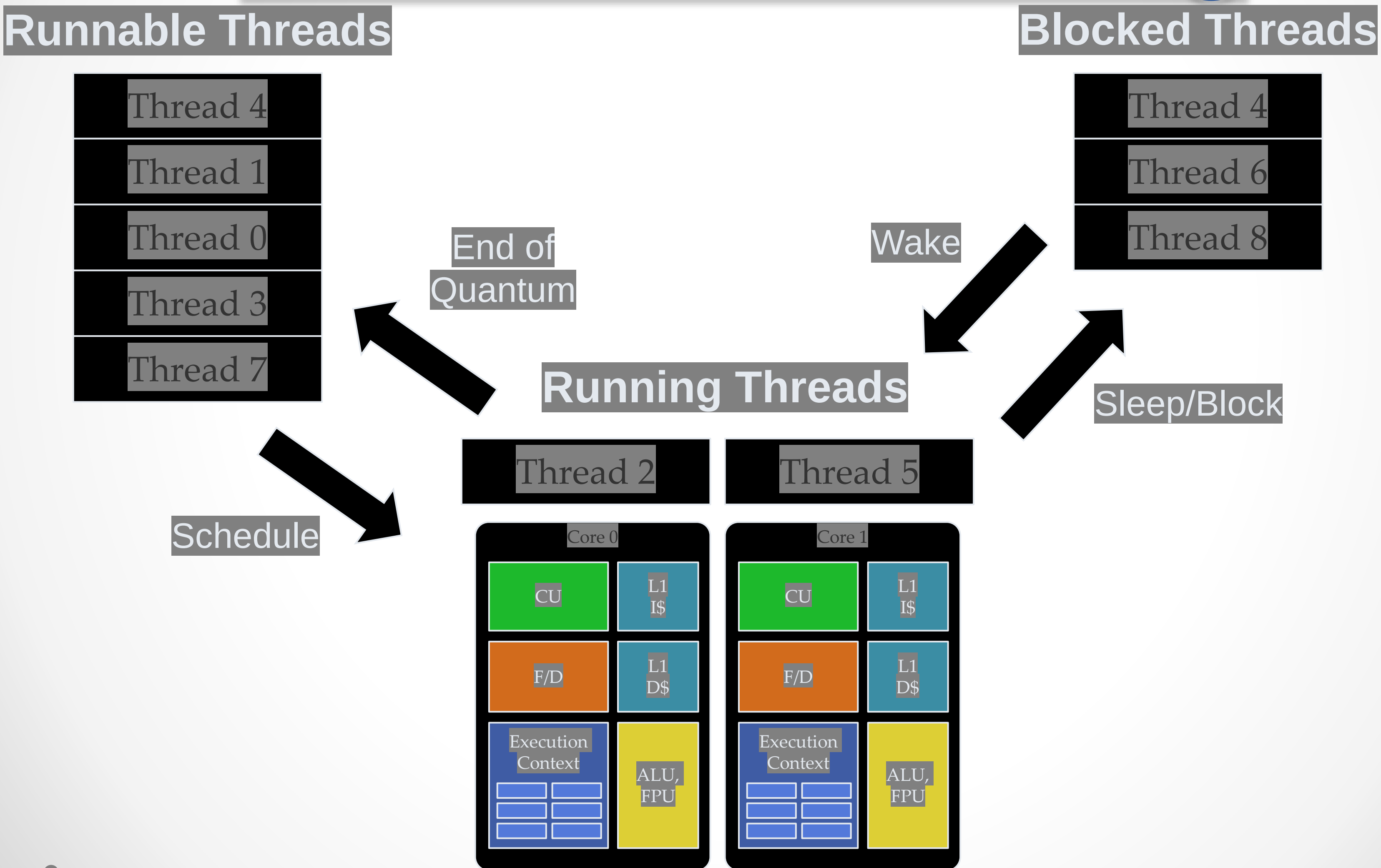
- Basic thread state machine*
  - **Runnable**
    - Thread is able to run, but is waiting to be assigned a time slice on a core
  - **Running**
    - Thread is actively running on a core
  - **Sleeping/Blocked/Waiting**
    - Thread is waiting for an event, timer, mutex lock, etc.
    - In this case, we say the thread is **blocked**

*Real operating systems have a few more states

Thread Scheduling

- A few examples of **blocking calls**
  - **`usleep()`** puts the thread to sleep for specified number of microseconds
  - **`read()`** puts the thread to sleep until the requested data has been read by the file system
  - A thread can also be put to sleep while waiting for a **mutex lock** (more on this later!)
- Sleeping threads are **woken up** by kernel when the **resource** becomes available or **timer** expires

Thread Scheduling



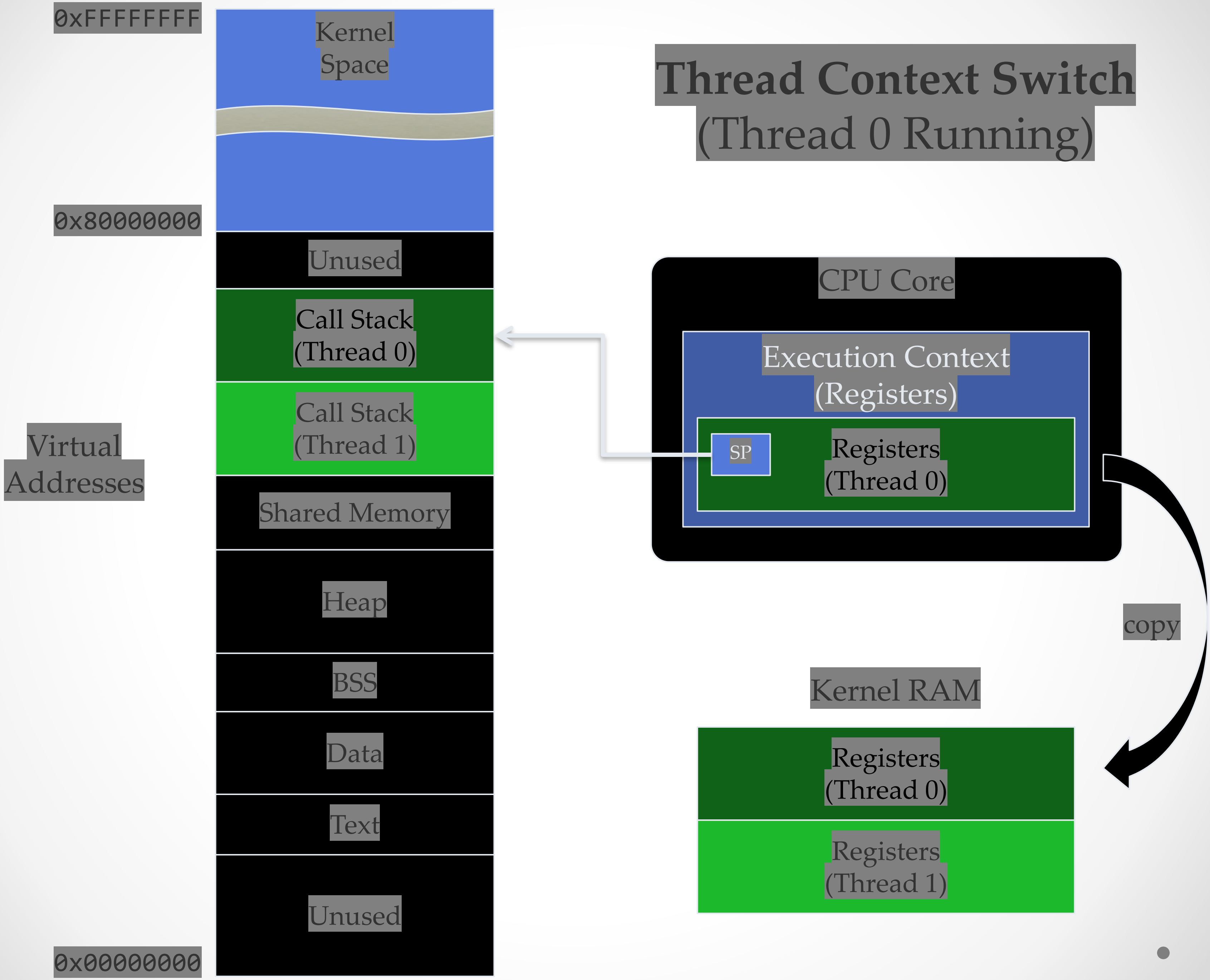
Thread Scheduling

- When a thread transitions from **Running** to
  - **Runnable** or
  - **Blocked**

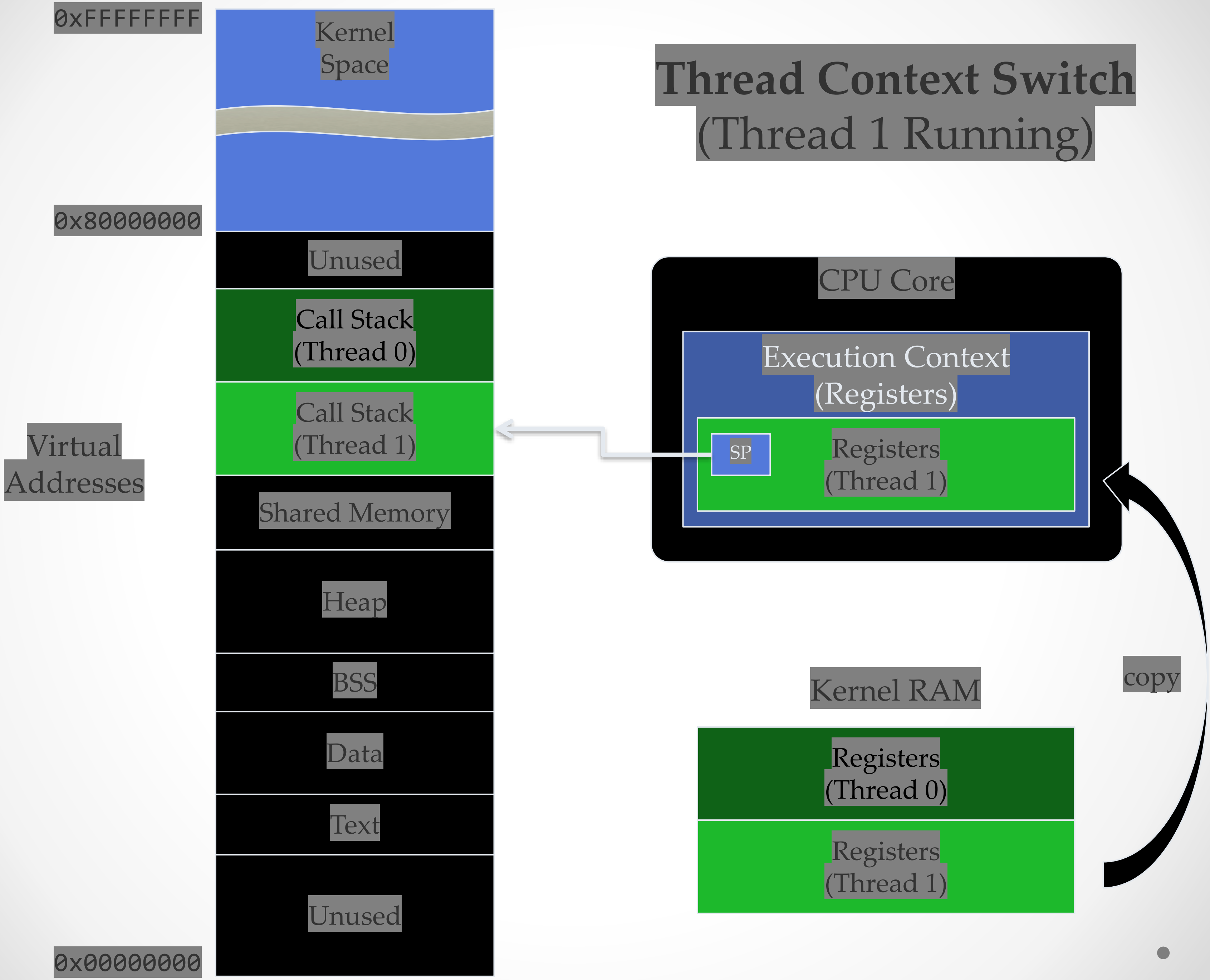
… and another thread is scheduled on a core, we call this a **context switch**

- Context switches also happen when a thread makes a **kernel call**
- Context switch involves **saving** the registers of the outgoing thread; **restoring** registers of incoming thread

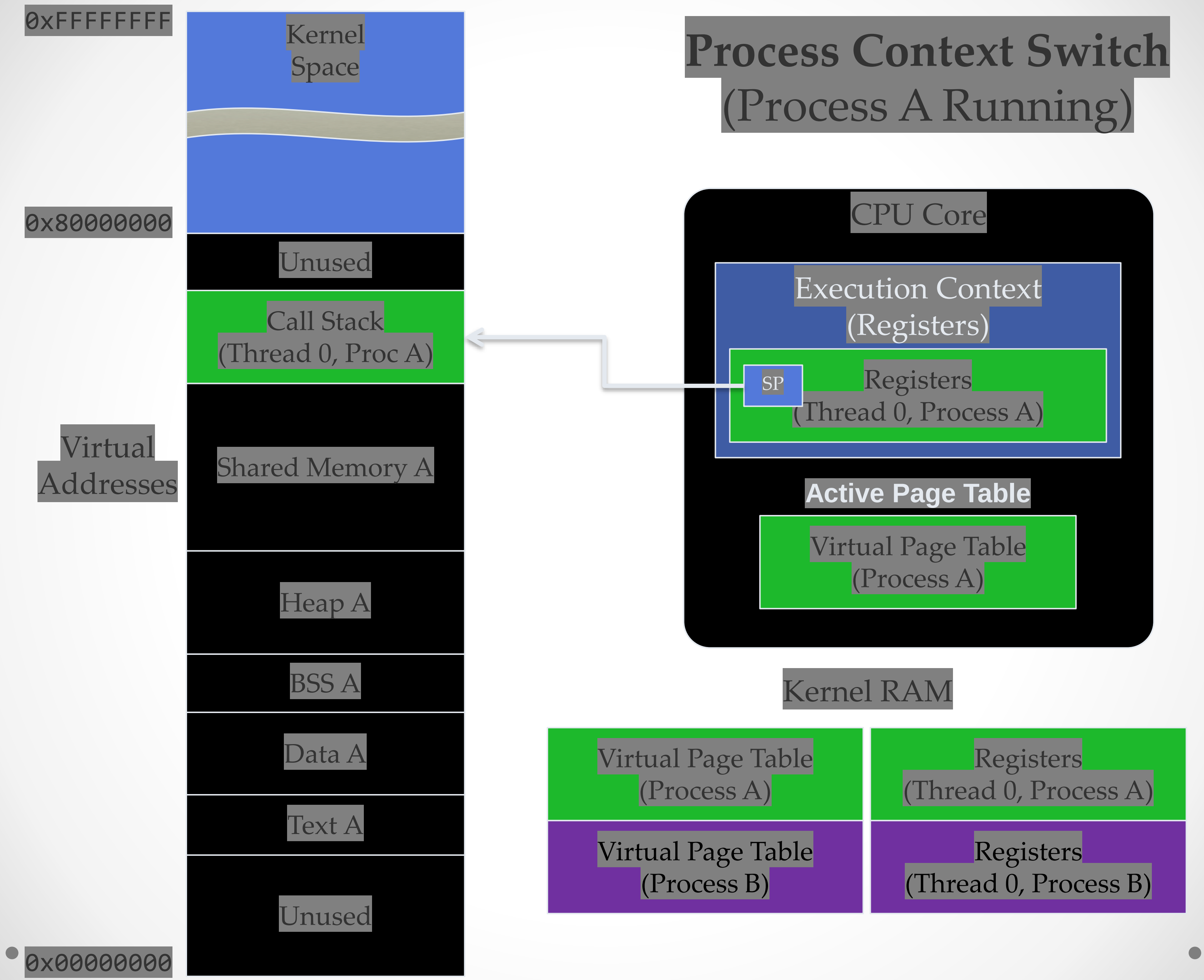
Thread Context Switch (Thread 0 Running)



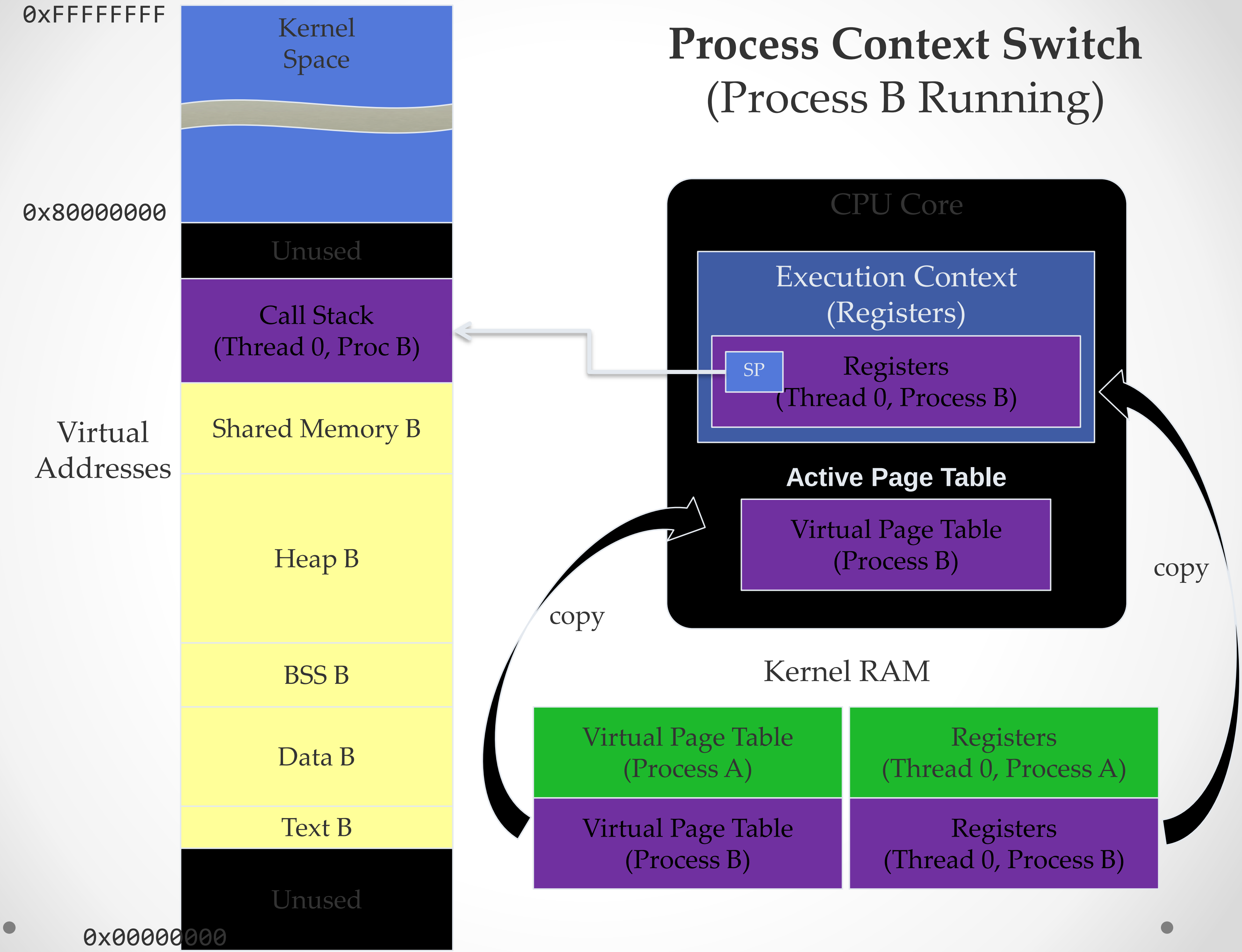
Thread Context Switch
(Thread 1 Running)



Process Context Switch (Process A Running)



Process Context Switch
(Process B Running)



Thread Scheduling

- Threads can be assigned **priorities**
    - Used to select threads from the **Runnable** state for time slices on available CPU core(s)
    - Higher-priority threads given **preference**
    - Can lead to **starvation** of lower-priority threads
- Process's **nice** value can also affect the effective priorities of its threads

Thread Scheduling

- Case study: **Linux** thread scheduling
  - Threads can be given a **scope**
    - PTHREAD_SCOPE_PROCESS versus PTHREAD_SCOPE_SYSTEM
    - Each SYSTEM thread gets equal share of CPU time
    - Each group of threads within a single process at PROCESS scope gets a share of CPU equal to one SYSTEM thread
  - **Priorities** work like this:
    - When a thread is runnable, and ***no other thread*** within the process has a ***higher*** priority, it will be scheduled
    - Threads with ***equal*** priority round-robin across available cores
  - **Starvation** of lower-priority threads unless all higher-priority threads **block**

Thread Scheduling

https://www.microsoftpressstore.com/articles/article.aspx?p=2233328&seqNum=7

- Case study: **Windows** thread scheduling
  - Threads can have one of 32 priorities
    - Separate queue for each priority
    - Highest-priority runnable thread always runs, with caveat…
    - … threads can be assigned processor affinity
  - Each thread receives a **time quantum**, defined by various factors (system config, foreground/background process)
    - A thread **may not complete** its quantum, if a higher-priority thread becomes runnable during the quantum
  - Unlike Linux, every thread is given equal weight
    - No consideration of to which process each thread belongs