Principled Programming

Introduction to Coding in Any Imperative Language

Tim Teitelbaum

Emeritus Professor

Department of Computer Science

Cornell University

Prerequisites

Prerequisite notions used for the rest of the book are presented here:

- Programming Concepts
- Programming Constructs
- English Conventions
- Hardware/OS Concepts [optional]

Programming Concepts

algorithm. An algorithm is a method for solving a problem, or performing a task.

program. A program is an algorithm written down in a programming language.

programming language. A programming language is a system of notation for programs that can be executed by a computer.

computer. A computer is a device for executing programs written in a programming language. A computer has a processor and a memory.

processor. A processor is a device that can obey the instructions of a machine-code program.

memory. A memory is a device that stores both machine code and values.

machine code. Machine code is a low-level programming language specific to a particular processor.

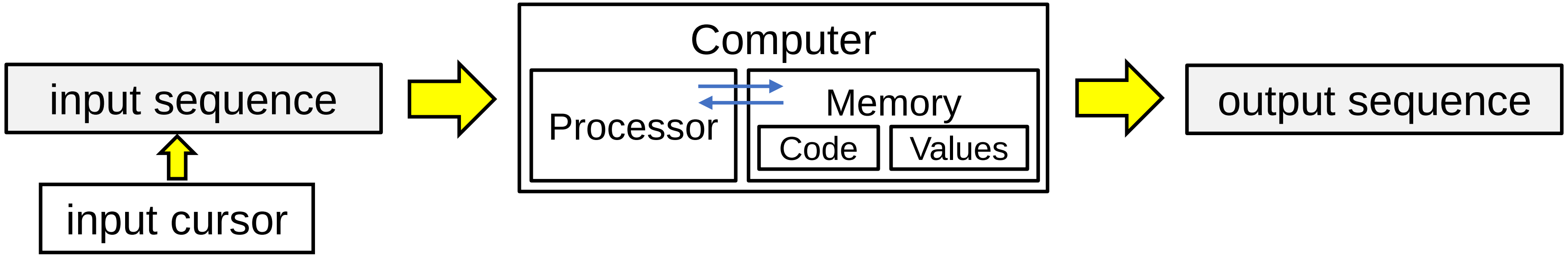
execution. To execute a program is to perform the steps it dictates. Execution is also known as running the program. Execution of a machine-code program follows the fetch-execute cycle, whereby the processor repeatedly performs the two steps:

- **Fetch** the next machine-code instruction from memory.
- **Execute** that instruction.

Analogously, execution of a program written in a high-level language repeatedly performs two steps:

- **Fetch** the next statement.
- **Execute** that statement.

environment. A program is executed by a computer in an environment that includes its external data, i.e., its input data and its output data:



external data. Input data are a linear sequence of characters, with a distinguished point denoted by the **input cursor** that indicates the next character to be input. **Output data** are a linear sequence of characters, to which the program can append at the end.

compiler. A compiler is a program that can translate a program written in a high-level programming language, e.g., Java, into an equivalent program written in a low-level programming language, e.g., machine code for the Intel x86 family of processors.

interpreter. An interpreter is a program that can execute a program written in a high-level language without first using a compiler to translate it to machine code.

Concepts

value. A value is an entity that is manipulated by a program. Values have types.

type. The type of a value is a categorization that determines how the value can be used in computation.

variable. A *variable* is a named memory location that can contain a value of a particular type. A *variable* is depicted by a box, prefixed by its *name*, and containing its *value*.

name | *value* |

assignment. Assignment is the act of storing a value in a variable, thereby overwriting its previous contents.

Concepts

statement. A *statement* is a programming language construct whose execution has an effect on the state of execution.

state. The state of a program's execution consists of a location in its code, the values of its variables, the text in its input and output data, and the position of its input cursor.

effect. An effect is a change in the state of a program's execution. The program is said to transition from one state to another.

location. A location in code is the statement being executed, and the ordered list of method call sites whose invocations are not yet complete.

Concepts

expression. An *expression* is a programming language construct whose evaluation yields a *value*. An **arithmetic expression** is an *expression* whose value has a numeric *type*.

condition. A *condition* is an *expression* whose value is logical rather than numeric, i.e., either **true** or **false**. Such values are also known as *type* **boolean**.

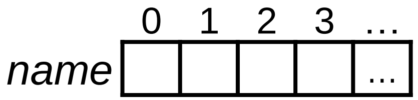
evaluation. To evaluate an expression is to perform its *operations* on its *operands*, where these are specific to the programming-language constructs supported.

declaration. A *declaration* is a programming language construct whose execution has the effect of creating a variable. The *name* of a variable has a scope, and the variable has a lifetime.

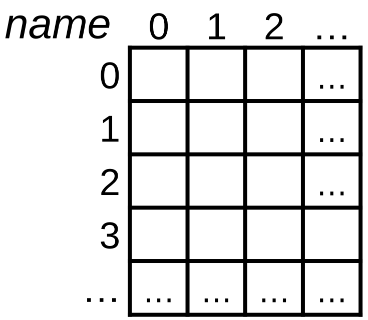
scope. The scope of a variable is the portion of a program's text where the variable's *name* is meaningful.

lifetime. The lifetime of a variable is the time interval within a program's execution during which the variable exists.

1-D array. A one-dimensional array is a linear sequence of variables indexed by consecutive integers, starting at 0.



2-D array. A two-dimensional array is a rectangular arrangement of variables indexed by pairs of integers, the row and column, each of which starts at 0.



scalar. A scalar *variable* is a *variable* that is not an array.

definition. A *definition* is a programming language construct that creates a *method* or a *class*.

method / **procedure** / **function**. A method is a *named*, parameterized sequence of *declarations* and *statements* that can executed by **invoking** (or **calling**) it from a *statement* or an *expression*. Methods have **return-types**. If the return type is **void**, the invocation only has effect, and can only appear in a *statement*. If the return-type is non-**void**, the method is known as a function, its invocation yields a value of the given *type*, and can appear in appear in an *expression* or *statement*. The return value of a function that is invoked as a *statement* is discarded. Methods are also known as procedures.

class. A class is a group of related *declarations* and *definitions*.

Programming Constructs

Constructs
Statements
Variables
Expressions
Types
Declarations and Definitions
Arguments & Parameters
Comments

Constructs
Statements
Variables
Expressions
Types
Declarations and Definitions
Arguments & Parameters
Comments

```
import java.util.Scanner;
class boilerplate {
   static Scanner in = new Scanner(System.in);
   static void main() {
      /* Output the Integer Square Root of an
         integer input. */
         /* Obtain an integer n≥0 from the user. */
            int n = in.nextInt();
         /* Given n≥0, output the Integer Square
            Root of n. */
            /* Let r be the integer part of the
               square root of n≥0. */
               int r = 0;
               while ( (r+1)*(r+1) <= n ) r++;
            System.out.println( r );
   } /* main */
} /* boilerplate */
```

Constructs
☞ Statements
Variables
Expressions
Types
Declarations and Definitions
Arguments & Parameters
Comments

```
import java.util.Scanner;
class boilerplate {
   static Scanner in = new Scanner(System.in);
   static void main() {
      /* Output the Integer Square Root of an
         integer input. */
         /* Obtain an integer n≥0 from the user. */
            int n = in.nextInt();
         /* Given n≥0, output the Integer Square
            Root of n. */
            /* Let r be the integer part of the
               square root of n≥0. */
               int r = 0;
               while ( (r+1)*(r+1) <= n ) r++;
            System.out.println( r );
   } /* main */
} /* boilerplate */
```

Constructs
Statements
☞ Variables
Expressions
Types
Declarations and Definitions
Arguments & Parameters
Comments

```
import java.util.Scanner;
class boilerplate {
   static Scanner in = new Scanner(System.in);
   static void main() {
      /* Output the Integer Square Root of an
         integer input. */
         /* Obtain an integer n≥0 from the user. */
            int n = in.nextInt();
         /* Given n≥0, output the Integer Square
            Root of n. */
            /* Let r be the integer part of the
               square root of n≥0. */
               int r = 0;
               while ( (r+1)*(r+1) <= n ) r++;
            System.out.println( r );
   } /* main */
} /* boilerplate */
```

Constructs
Statements
Variables
☞ Expressions
Types
Declarations and Definitions
Arguments & Parameters
Comments

```
import java.util.Scanner;
class boilerplate {
   static Scanner in = new Scanner(System.in);
   static void main() {
      /* Output the Integer Square Root of an
         integer input. */
         /* Obtain an integer n≥0 from the user. */
            int n = in.nextInt();
         /* Given n≥0, output the Integer Square
            Root of n. */
            /* Let r be the integer part of the
               square root of n≥0. */
               int r = 0;
               while ( (r+1)*(r+1) <= n ) r++;
            System.out.println( r );
   } /* main */
} /* boilerplate */
```

Constructs
Statements
Variables
Expressions
☞ Types
Declarations and Definitions
Arguments & Parameters
Comments

```
import java.util.Scanner;
class boilerplate {
   static Scanner in = new Scanner(System.in);
   static void main() {
      /* Output the Integer Square Root of an
         integer input. */
         /* Obtain an integer n≥0 from the user. */
            int n = in.nextInt();
         /* Given n≥0, output the Integer Square
            Root of n. */
            /* Let r be the integer part of the
               square root of n≥0. */
               int r = 0;
               while ( (r+1)*(r+1) <= n ) r++;
            System.out.println( r );
   } /* main */
} /* boilerplate */
```

Constructs
Statements
Variables
Expressions
Types
☞ Declarations and Definitions
Arguments & Parameters
Comments

```
import java.util.Scanner;
class boilerplate {
   static Scanner in = new Scanner(System.in);
   static void main() {
      /* Output the Integer Square Root of an
         integer input. */
         /* Obtain an integer n≥0 from the user. */
            int n = in.nextInt();
         /* Given n≥0, output the Integer Square
            Root of n. */
            /* Let r be the integer part of the
               square root of n≥0. */
               int r = 0;
               while ( (r+1)*(r+1) <= n ) r++;
            System.out.println( r );
   } /* main */
} /* boilerplate */
```

Constructs
Statements
Variables
Expressions
Types
Declarations ☞ Definitions
Arguments & Parameters
Comments

```
import java.util.Scanner;
class boilerplate {
   static Scanner in = new Scanner(System.in);
   static void main() {
      /* Output the Integer Square Root of an
         integer input. */
         /* Obtain an integer n≥0 from the user. */
            int n = in.nextInt();
         /* Given n≥0, output the Integer Square
            Root of n. */
            /* Let r be the integer part of the
               square root of n≥0. */
               int r = 0;
               while ( (r+1)*(r+1) <= n ) r++;
            System.out.println( r );
   } /* main */
} /* boilerplate */
```

Constructs
Statements
Variables
Expressions
Types
Declarations and Definitions
☞ Arguments & Parameters
Comments

```
import java.util.Scanner;
class boilerplate {
   static Scanner in = new Scanner(System.in);
   static void main() {
      /* Output the Integer Square Root of an
         integer input. */
         /* Obtain an integer n≥0 from the user. */
            int n = in.nextInt();
         /* Given n≥0, output the Integer Square
            Root of n. */
            /* Let r be the integer part of the
               square root of n≥0. */
               int r = 0;
               while ( (r+1)*(r+1) <= n ) r++;
            System.out.println( r );
   } /* main */
} /* boilerplate */
```

Constructs
Statements
Variables
Expressions
Types
Declarations and Definitions
Arguments & Parameters
☞ Comments

```
import java.util.Scanner;
class boilerplate {
   static Scanner in = new Scanner(System.in);
   static void main() {
      /* Output the Integer Square Root of an
         integer input. */
         /* Obtain an integer n≥0 from the user. */
            int n = in.nextInt();
         /* Given n≥0, output the Integer Square
            Root of n. */
            /* Let r be the integer part of the
               square root of n≥0. */
               int r = 0;
               while ( (r+1)*(r+1) <= n ) r++;
            System.out.println( r );
   } /* main */
} /* boilerplate */
```

A statement is evaluated for its effect.

```
variable = expression;
```

Meaning: Assign the value of *expression* to the *variable*.

```
variable++;
```

Meaning: Short for *variable*=*variable*+1; and called *auto-increment*.

```
variable--;
```

Meaning: Short for *variable*=*variable*-1; and called *auto-decrement*.

if (*condition*) *statement*$_1$ else *statement*$_2$

Meaning: Execute *statement*$_1$ if the value of the *condition* is **true**,
otherwise execute *statement*$_2$.

if (*condition*) *statement*

Meaning: Execute *statement* if the value of the *condition* is **true**.

```
while ( condition ) statement
```

Meaning: Repeatedly execute *statement* provided the *condition* is **true** before each execution. A **while-statement** is called a *loop*, and its constituent statement is called its *body*. Executing the *body* zero or more times is called *iterating*.

```
for ( initialize; condition; update ) statement
```

Meaning: Equivalent to:

```
initialize;
while (condition) {
    statement
    update
    }
```

where *initialize* is one of

```
name = expression
type name = expression
```

where *update* is one of

```
name = expression
name++
name--
```

```
block
```

Meaning: Execute the *block*, which groups *declarations* and *statements*, and permits them to be used in a context where only a single *statement* is otherwise allowed, e.g., in an **if**-statement, a **while**-statement, or a **for**-statement.

where a *block* has the form:

```
{ declarations-and-statements }
```

Meaning: Execute each declaration and/or statement, in sequence. Variables declared in a block go out of existence each time execution of the block completes.

```
System.out.println( expression );
```

Meaning: Convert the value of *expression* to a `String` (if necessary), append it to the output data, and advance to the beginning of the next line in the output data.

```
System.out.print ( expression );
```

Meaning: Convert the value of *expression* to a `String` (if necessary), append it to the output data, and remain on the same line in the output data.

```
System.out.println( );
```

Meaning: Advance to the beginning of the next line in the output data.

```
name( arguments );
```

Meaning: Invoke the *named* **void** method with the values of *arguments*, which is a comma-separated list of *expressions*. Invoking a method with a list of arguments has the effect of:

- evaluating each *argument expression*,
- declaring new *variables* for the method's *parameters*,
- assigning the argument values to the corresponding parameters,
- evaluating the *block* of the method, and
- returning to the invocation site, either by execution of a **return** statement, or by completing execution of the method's body.

```
return;
```

Meaning: Return to the method invocation site. This form of **return** *statement* is only permitted in a method of type **void**.

```
return expression;
```

Meaning: Return to the method invocation site with the value of *expression*. If the method has non-**void** *type* t, *expression* must have *type* t. This form of **return** statement is not permitted in a method of type **void**.

A variable is a location in memory that can contain a value.

name

Meaning: The *variable* with the given *name*.

class-name.name

Meaning: The *named* variable that is declared in class *class-name*, e.g., `Integer`.MAX_VALUE. Also, a *named* method of *class-name,* e.g., `Math.sqrt`.

`name[ expression ]`

Meaning: The value of *expression* is known as an index, and the *named* 1-D array is known as a *subscripted variable*. Let k be the value of *expression*. The variable denoted by *name*[*expression*] is the k^{th} variable of the array, starting at the 0^{th} variable. If k is negative, or is not less than the length of the array, a runtime error is triggered.

name[$expression_1$][$expression_2$]

Meaning: The named variable is a 2-D array, and the meaning is similar to the 1-D case. $Expression_1$ and $expression_2$, known as the row and column indices, are required to be less than the height and width of the named array, respectively.

Constructs

expressions

A expression is *e*valuated to obtain its value.

- Constants
- Primitives
- Binary Operations
- Unary Operations
- Grouping

`0`, `1`, `2`, …, `-1, -2`, …	(type **int**)
`0L`, `1L`, `2L`, …, `-1L, -2L`, …	(type **long**)
`6.0221409f+23`, …	(type **float**)
`0.0`, `3.14159`, `6.0221409e+23`, …	(type **double**)
true, **false**	(type **boolean**)
`'a'`, `'b'`, `'c'`, …, '\u0000'	(type **char**)
"*characters*"	(type String)

variable

Meaning: The value contained in the *variable*.

`name( arguments )`

Meaning: The value returned by an invocation of the named **non-void** method with the values of the *arguments*. The final statement executed by the method must be a **return**- statement, which provides the value for the method invocation. (See method invocation under *statements*.)

```
in.nextInt()
```

Meaning: The **int** value returned by invoking the method `in.nextInt()`. The value returned is the base-10 integer in the input data at the position of the input cursor, converted to its binary fixed-point form. Invocation has the effect of advancing the input cursor beyond the integer that has been read. Variable `in` is assumed to have been initialized by:

```
Scanner in = new Scanner(System.in);
```

earlier in the code.

new *type*[*expression*]

Meaning: Create a 1-dimensional array of variables of the given *type*, whose length is given by the value of *expression*. The variables of the array are known as its *elements*, and are indexed by nonnegative integers, starting at 0.

new *type*[$expression_1$][$expression_2$]

Meaning: Create a 2-dimensional array of variables of the given *type*, whose height and width are given by the values of $expression_1$ and $expression_2$, respectively. The variables of the array are known as its *elements*, and are indexed by pairs of nonnegative integers, starting at 0.

`+, -, *, /, %` (arithmetic)

Meaning: Arithmetic operators "`+`", "`-`", and "`*`" (addition, subtraction, and multiplication) are standard. Operation "`/`" (division) truncates the fractional part of the quotient if both operands are **`int`** or **`long`**, and includes the fractional part otherwise. Operation "`%`" (modulus) is the integer remainder after integer division.

`<, <=, >, >=, ==, !=` (relational)

Meaning: Relational operations “`<`”, “`<=`”, “`>`”, “`>=`”, “`==`”, and “`!=`” (less, less or equal, greater, greater or equal, equal, and not equal) are standard, and yield **`boolean`** results, i.e., **true** or **false**.

&&, || (**boolean**)

Meaning: Boolean operations "&&" and "||" are **and** and **or**, respectively.

`+`	(concatenation)

Meaning: The operation "`+`" is concatenation if one or both operands have type `String`. If one operands is arithmetic, the value is converted to its base-10 representation as a `String`.

`-`	(arithmetic)
`!`	(**`boolean`**)

Meaning: The unary operation "`-`" is arithmetic negation. The unary operation "`!`" is logical negation, i.e., "`!` *expression"* is **true** iff *expression* is **false**.

```
( expression )
```

A type is a characterization of a set of values.

`int`

Meaning: 32-bit, two’s-complement, fixed-point binary integer.

`long`

Meaning: 64-bit, two’s-complement, fixed-point binary integer.

`float`

Meaning: signed, 32-bit, floating-point number.

`double`

Meaning: signed, 64-bit, floating-point number.

`boolean`

Meaning: Either **true** or **false**.

`char`

Meaning: A single Unicode character

`String`

Meaning: A linear sequence of 0 or more Unicode characters.

`void`

Meaning: There are no values or variables of type **void**. A method defined with **void** return type can only be invoked as a *statement* for its effect.

type[ ]

Meaning: A value of type *type*[ ] signifies a 1-dimensional array of variables, each of which has type *type*.

type[ ][ ]

Meaning: A value of type *type*[ ][ ] signifies a 2-dimensional array of variables, each of which has type *type*.

A declaration creates named variables.

A *definition* defines a method or a class.

```
type name;
```

Meaning: Create a variable *name* of the given *type*. There are two kinds of variables.

Local variables. Declared in a method. Scope begins at *declaration*, and extends to end of the enclosing method. Lifetime of (each dynamic instance of) a local variable begins when the declaration is executed (in a new dynamic instance of the method), and ends when (that instance of) the method returns.

Class variables. Declared in a class, prefixed by the modifier **static**. Scope begins at *declaration*, and extends to end of the enclosing. The lifetime of a class variable begins when its declaration is executed, and lasts for the rest of program execution. The order in which classes are initiated is unspecified.

If *type* ends with [ ] or [ ] [ ], brackets can be moved to the right of *name*.

```
type name = expression;
```

Meaning: Create a variable *name* of the given *type*, and initialize it with the value of *expression*.

```
type name[] = { list-of-expressions };
```

Meaning: Create a variable *name* signifying a 1-D array of *type* elements that is initialized with values given by the comma-separated *list-of-expressions*.

```
static type name( parameters ) block
```

Meaning: Define a method with the given *name* and *parameters*. If type is **void**, the method can only be invoked as a *statement* for its effect. If type is non-**void**, the method can be invoked as an *expression* that computes a value. The block is called the *body* of the method. Methods of **void** type are referred to as *procedures*, and methods of non-**void** type are referred to as *functions*.

```
class name { declarations-and-methods }
```

Meaning: *Declarations-and-methods* is a list of intermixed *declaration* and *method definition* constructs. A *class* is a scope within which names of variables and methods are made accessible to the code therein. Outside the class, the names of variables, e.g., `v`, and methods, e.g., `m`, must be qualified by the class name, e.g., *class-name*.v and *class-name*.m.

Arguments are values that are provided to a method when it is invoked.

Parameters are variables created when a method is invoked that are initialized with the values of the corresponding arguments.

arguments is a comma-separated list of *expressions*

Meaning: Before entry to the method being invoked, each *argument expression* is evaluated.

parameters is a comma-separated list of *type-name* pairs

Meaning: On entry to the method being invoked, a variable is declared for each *type-name* pair. Each such variable, known as a *parameter*, has the given *type* and *name*, and is initialized with the value of the corresponding argument given in the method invocation. The scope of a *parameter* is the method definition in which it appears. The lifetime of (each dynamic instance of) a parameter begins on the invocation of (a new dynamic instance of) the method, and ends when (that instance of) the method returns.

Comments are ignored, but are essential to our methodology.

```
/* any-text */
```

Meaning: Ignored.

```
// any-text-to-end-of-line
```

Meaning: Ignored.

English conventions in comments

☞ **Write comments as an integral part of the coding process, not as afterthoughts.**

`Let` *`variable`* `be` *`text`*

Meaning: Set the *variable* (or *variables*) equal to the value(s) described by *text*.
Synonymous with “`Set` *`variable`* `equal to` *`text`*”.

`Given` text_1, text_2

Meaning: Provided that the state is as described by text_1, establish text_2.

name

Meaning: The *name* is either a local indeterminate used in the comment as a pronoun, or it is an actual program *variable*, in which case it either already exists, or is to be declared.

variable[$\mathtt{expression}_1$..$\mathtt{expression}_2$]

Meaning: The consecutive elements of the array *variable* with indices in the range $expression_1$ to $expression_2$, inclusive. When $expression_2$ is less than $expression_1$, the sub-array referred to is empty, i.e., contains no elements.

$\langle expression_1, expression_2 \rangle$

Meaning: A pair of values, considered as a single entity, consisting of the value of *expression*$_1$ and the value of *expression*$_2$.

`s.t.` *`text`*

Meaning: Such that *text*.

`i.e.,` *`text`*

Meaning: That is to say, *text*.

`e.g.,` *`text`*

Meaning: For example, *text*.

`iff` *`text`*

Meaning: if and only if *text*.

```
resp. text
```

Meaning: Respectively, *text*.

```
in situ
```

Meaning: In place, e.g., without an extra array of variables.

```
variable^expression
```

Meaning: *Variable* raised to the power given by the value of *expression*.

Additional concepts [optional]

- Hardware representations
- Operating System mechanisms

bit. A bit is the smallest unit of information in a computer. The word "bit" is both descriptive (as in, "a small quantity") and an acronym (**bi**nary **d**igit). A *bit* can be stored in a 2-state physical device, i.e., a switch that is either "on" or "off", "up" or "down", etc. By convention, the two possible states of a *bit* are known as 0 and 1.

byte. A byte is eight bits. Because each bit in a byte can independently be 0 or 1, a *byte* has 2^8=256 possible values.

byte-addressable memory. A byte-addressable memory consists of an ordered sequence of bytes (depicted by gray boxes) each of which has an individual numerical address.

address. An address is the name by which a byte in a memory is known. A memory's set of addresses is known as its address space.

word. A word is the unit of information conveyed to or from a memory in a single operation. Modern computers typically have 8-byte (64-bit) words. The locus of a memory-transfer operation is specified by the address of a single byte, but the transfer involves a whole word in the vicinity of that byte.

access time. The access time of a memory reference is the time required to convey a word of information to or from the memory.

RAM. A RAM is a physical device that implements a byte-addressable memory for which the access time is uniform and independent of the address. RAM is an acronym for Random Access Memory.

memory hierarchy. A stratification by size, access time, and price. Large memories are slow, but less expensive/byte. Smaller memories are faster, but more expensive/byte. Three strata: virtual, physical and cache. Efficiency gain from locality of memory accesses.

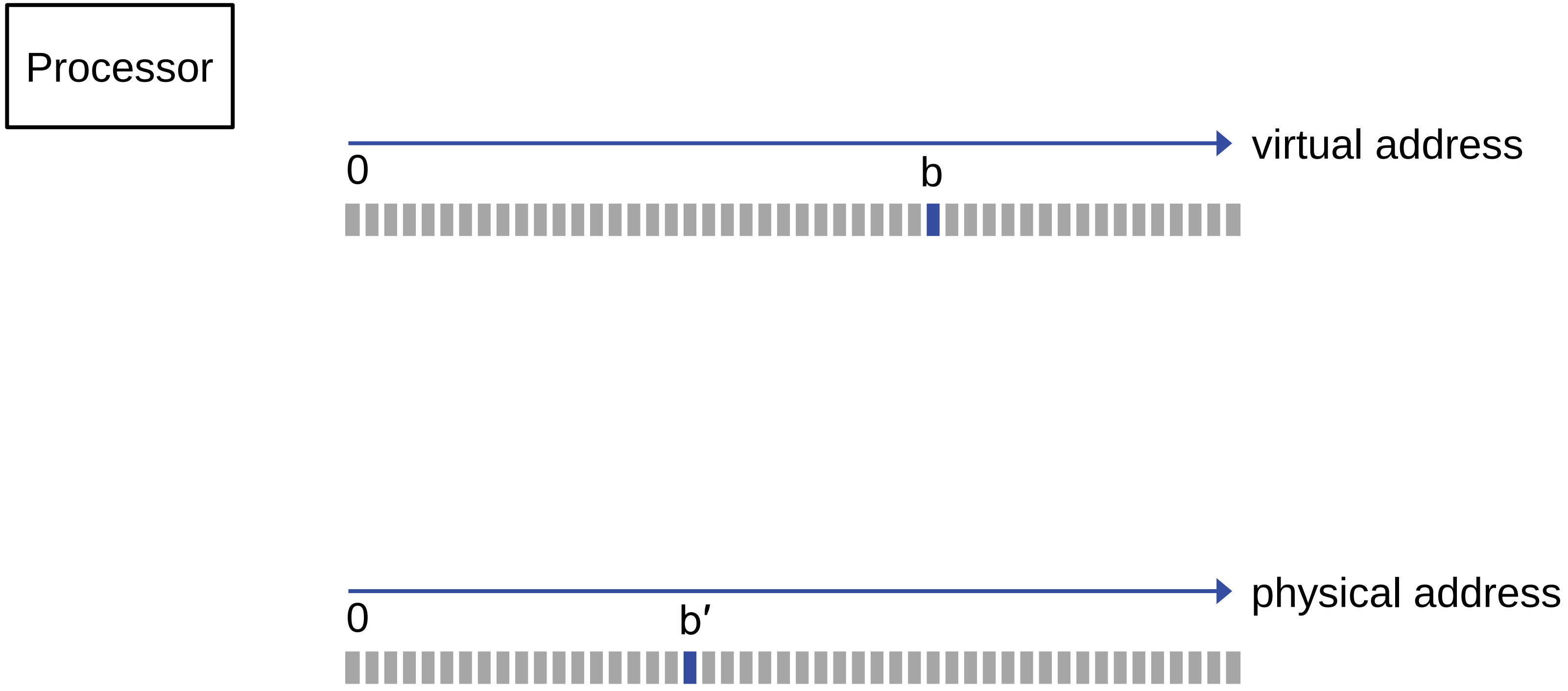
locality. Locality is a measure of the confinement of memory accesses to limited regions of an address space for extended periods of time. Locality allows bytes to temporarily reside in a smaller but faster stratum of the memory hierarchy.

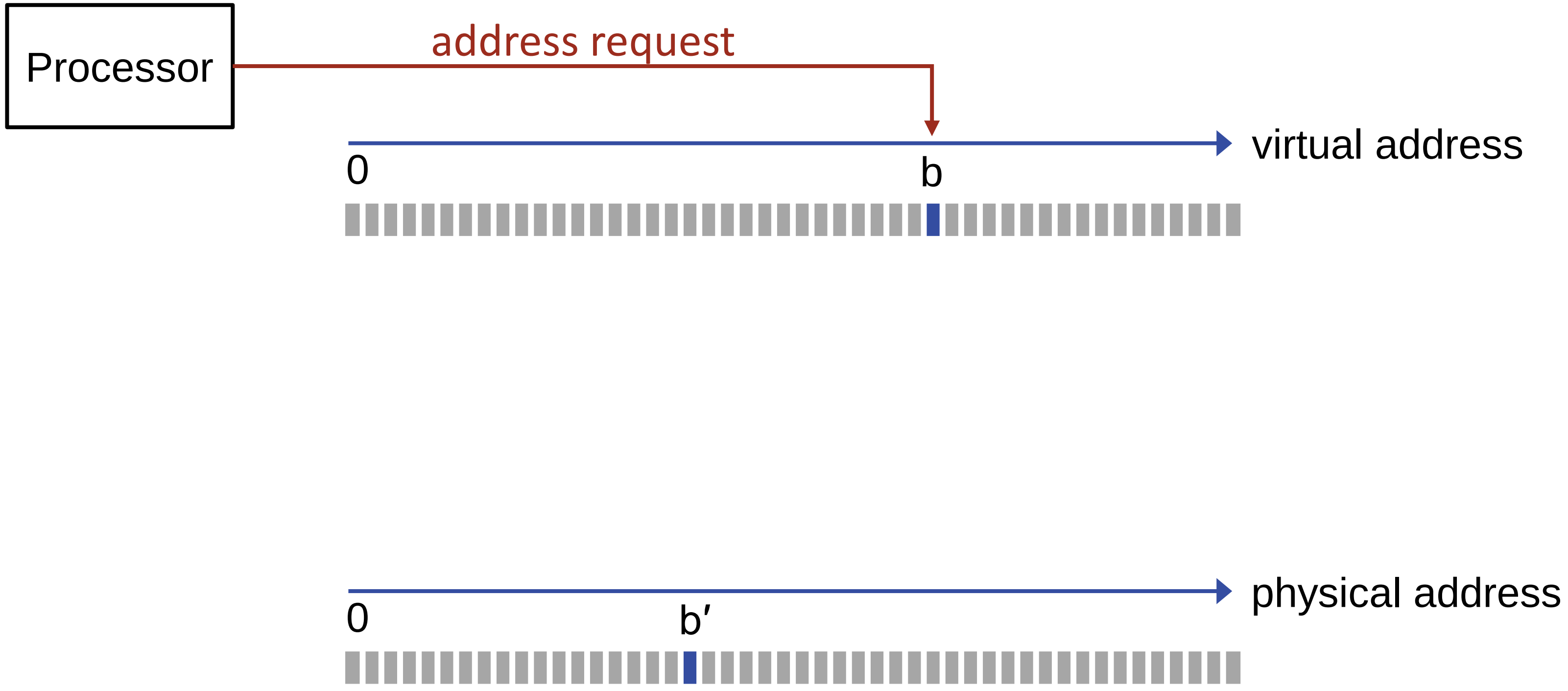
address translation. When a byte at address *b* in stratum s of one memory temporarily resides at address *b′* in another stratum *s′* of memory, references to *b* in *s* is address translated to *b′* in *s′*.

process. A process is a machine-code program in the midst of being executed. A computer may have multiple active processes at any given moment. Processes reference locations in virtual memory, where a **virtual address** typically corresponds to an offset in a region of disk memory reserved for the process. A disk may be an actual rotating device, or a solid-state facsimile of one. Disks have a large address space, and are inexpensive per byte.

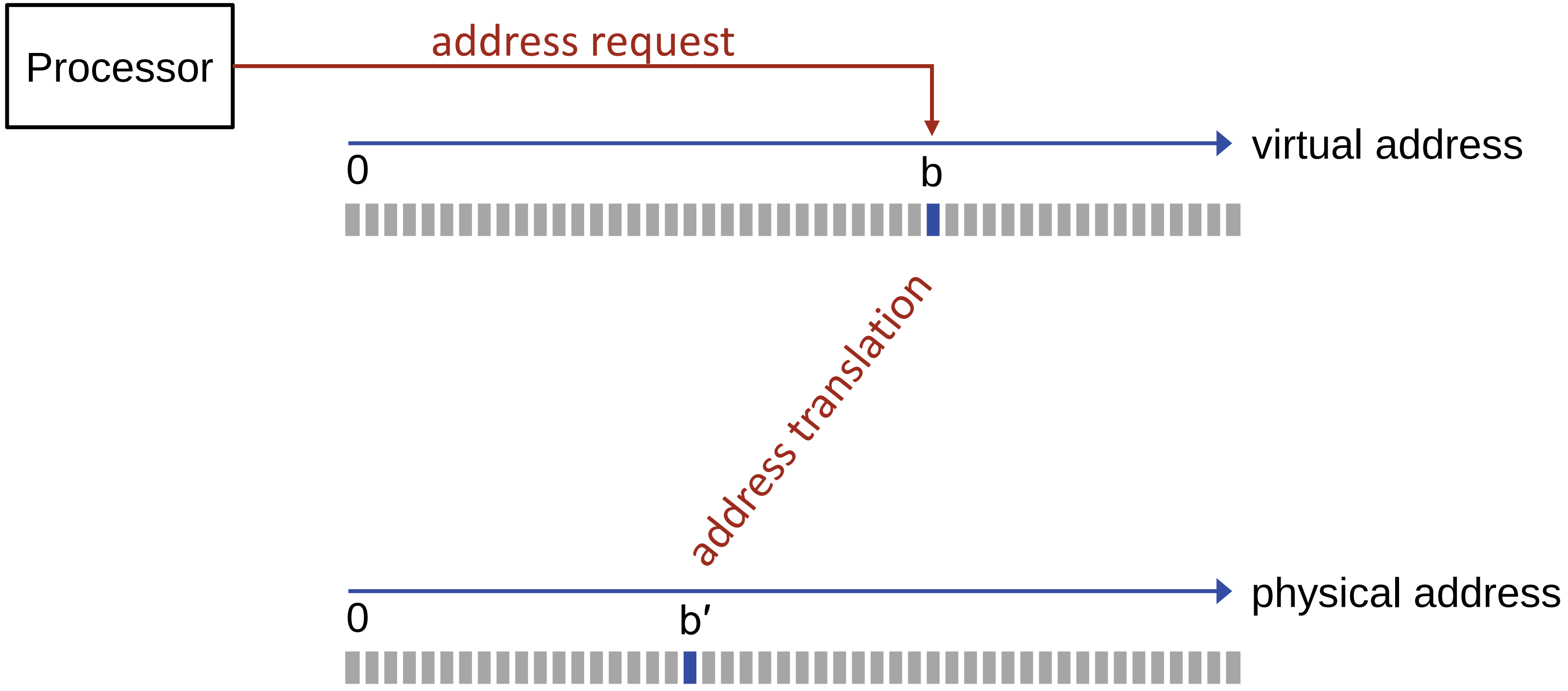
virtual memory. Processes run in a virtual address space, but processors execute programs in physical memory, where the correspondence between virtual and physical addresses is maintained dynamically by address translation.

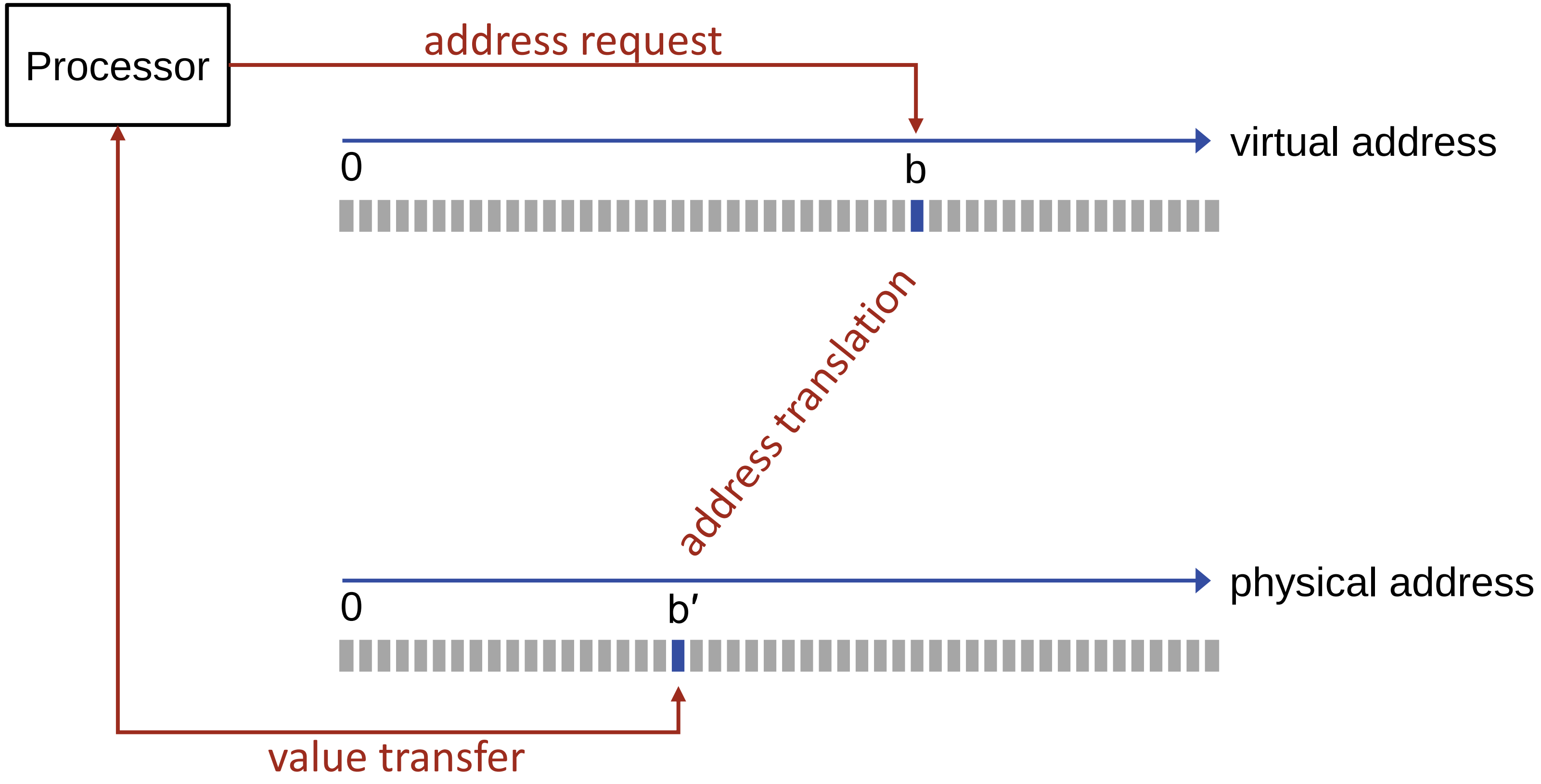
physical memory. The physical memory of a computer is a RAM that is shared by all active processes of the computer. Incremental installation of additional RAM increases the amount of virtual memory that can be mapped into physical memory at the same time, thereby reducing paging, and speeding execution.

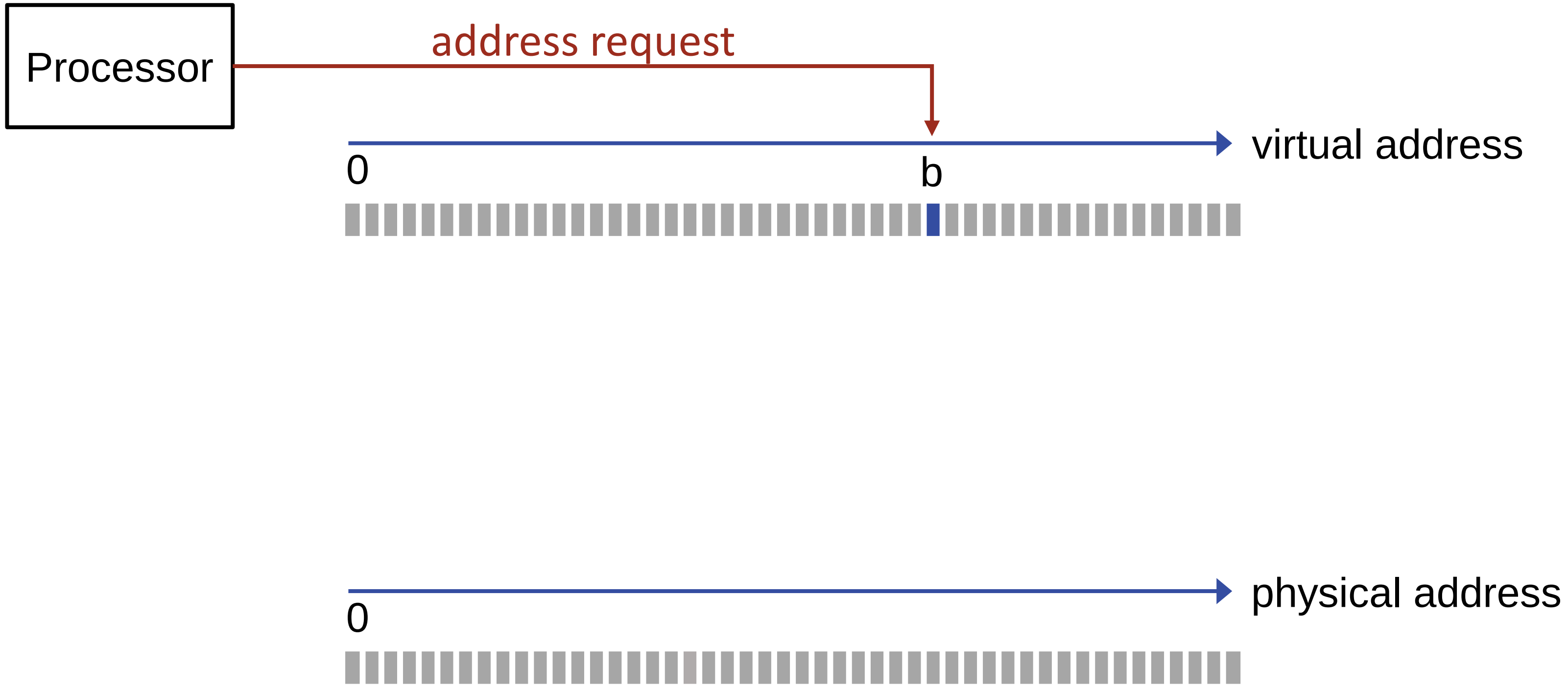

Processor
virtual address
0
b
physical address
0
b′

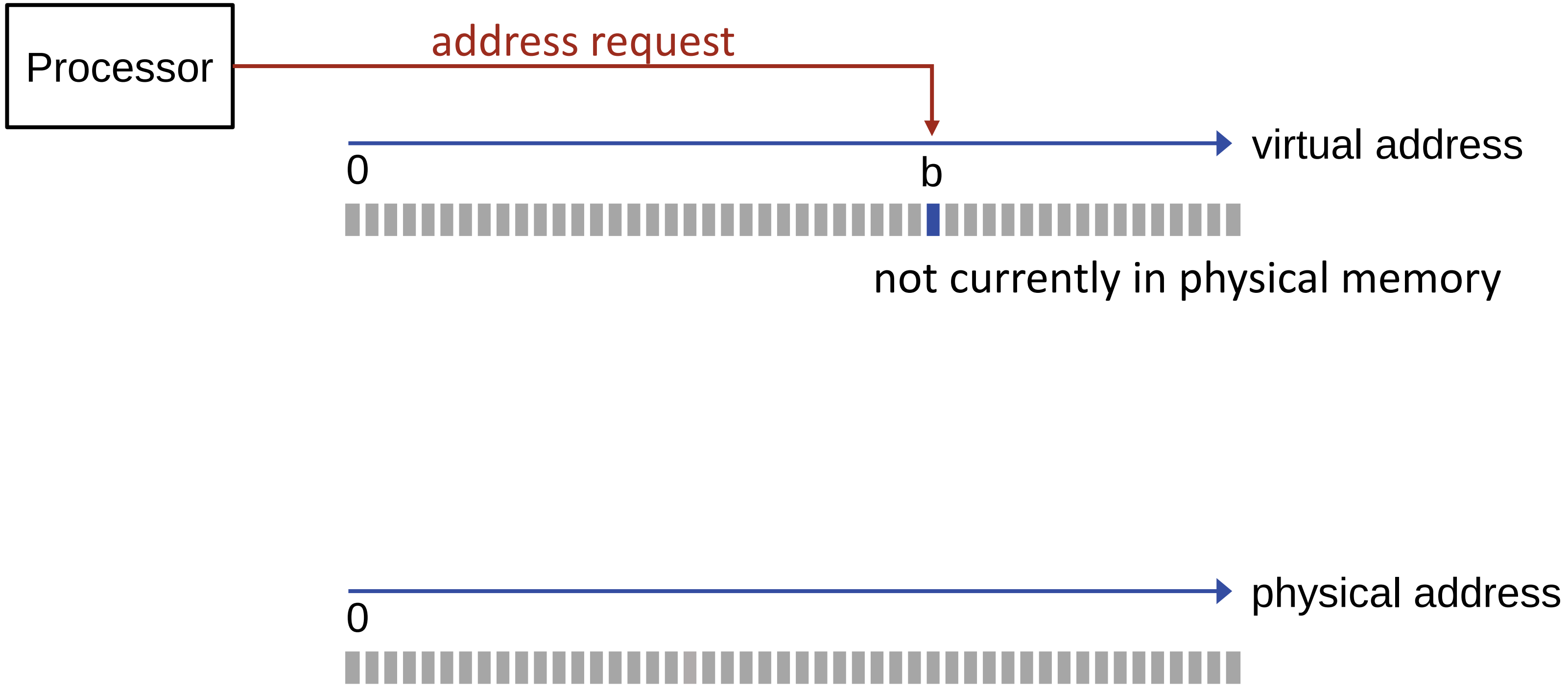

Processor
address request
0
b
virtual address
0
b′
physical address

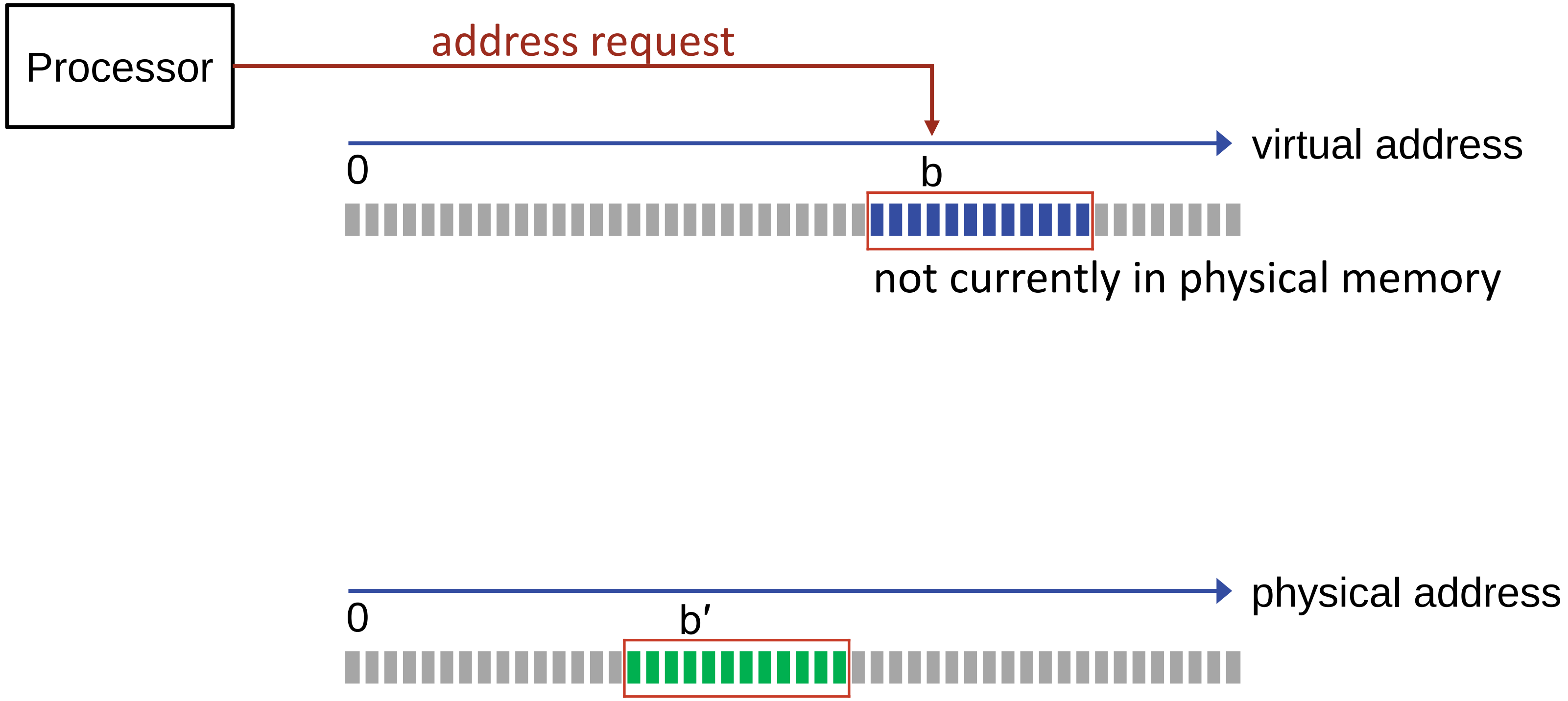
Concepts – Hardware & OS

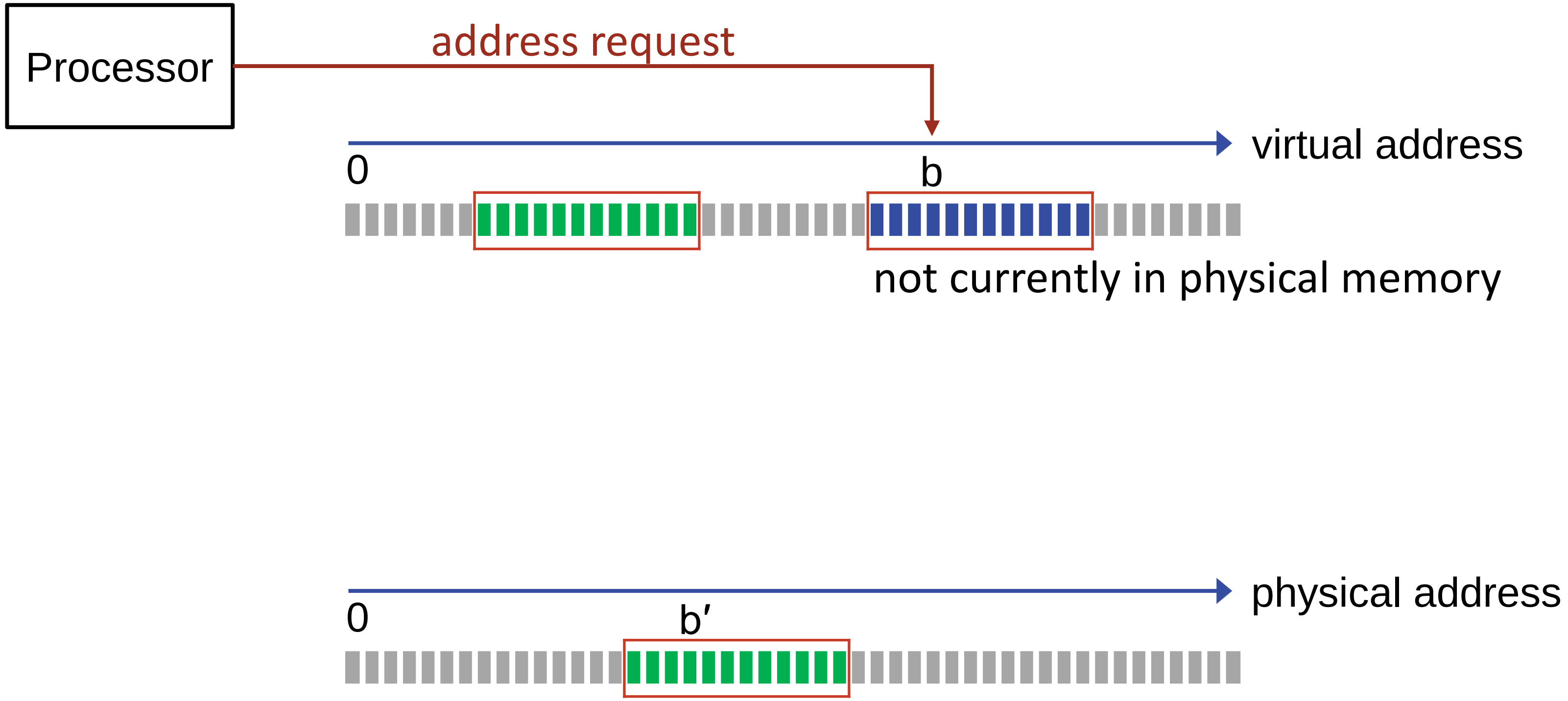



Processor
address request
virtual address
0
b
address translation
physical address
0
b′
value transfer

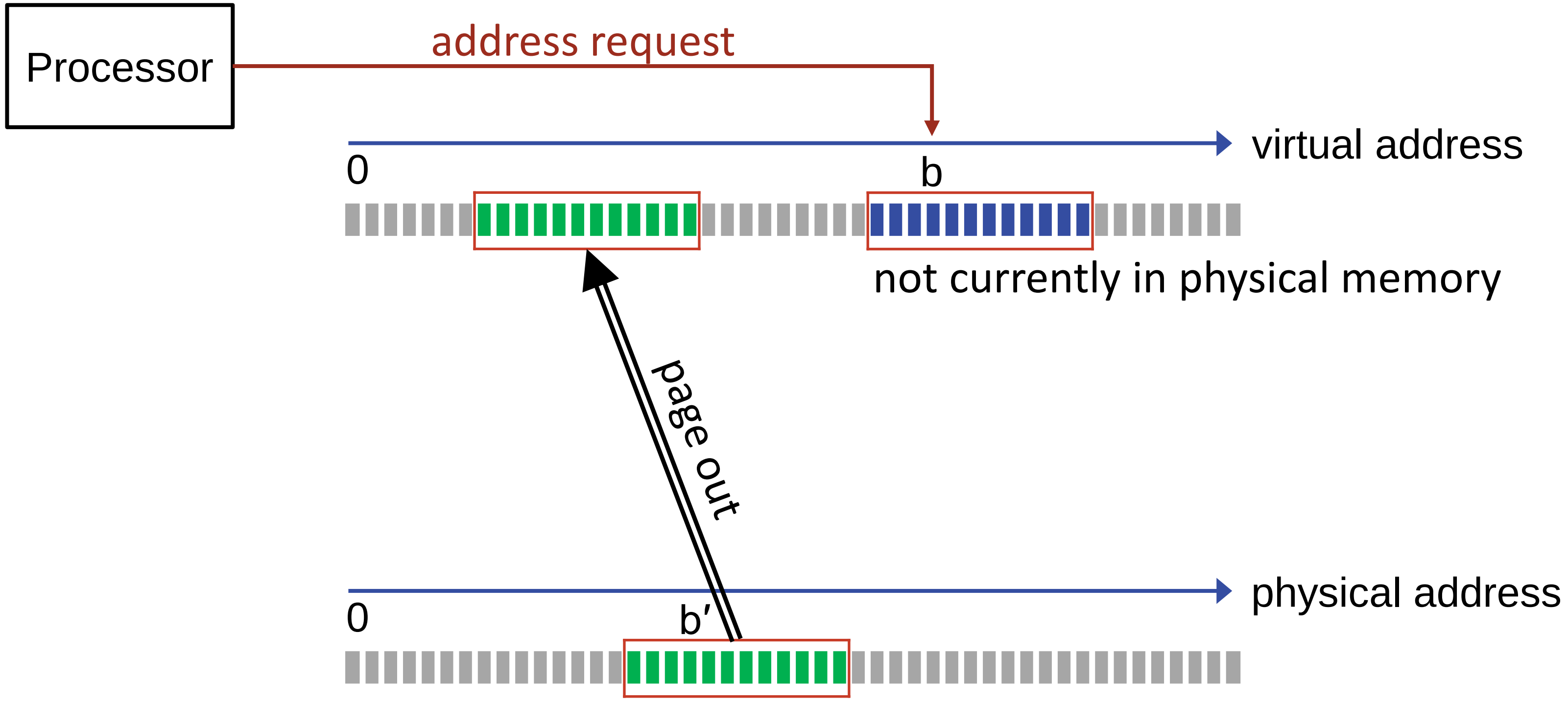

Processor
address request
virtual address
0
b
physical address
0

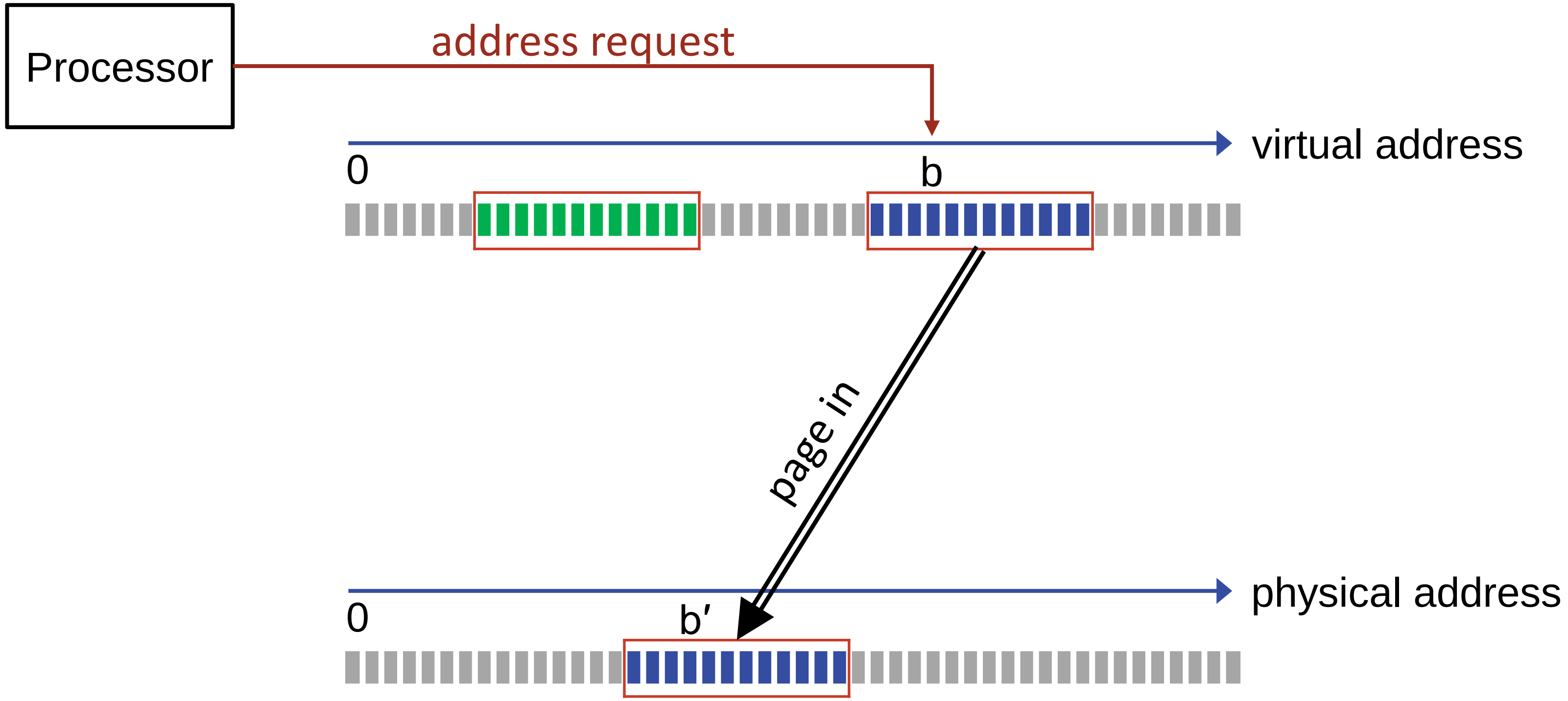

Processor
address request
0
b
virtual address
not currently in physical memory
0
physical address


Processor
address request
virtual address
0
b
not currently in physical memory
physical address
0
b′

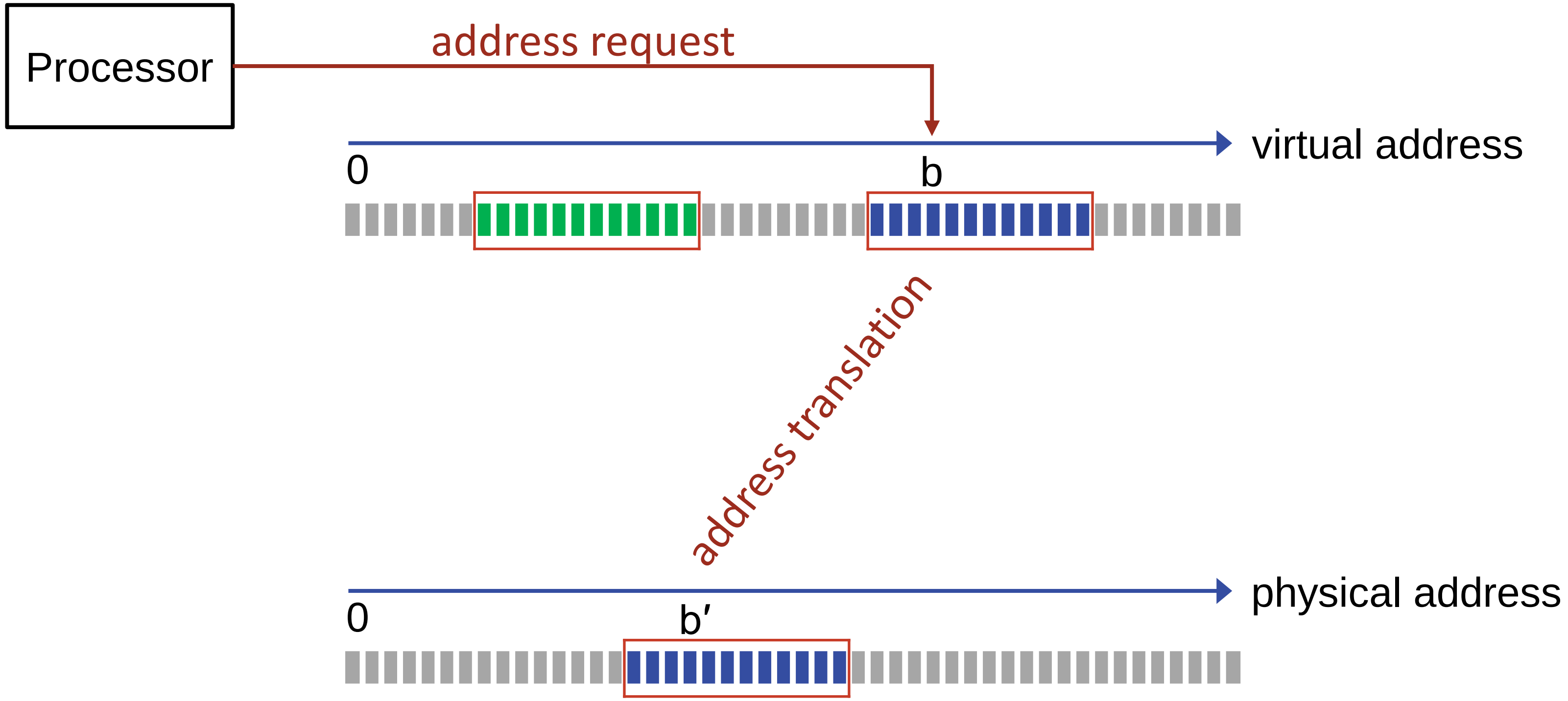

Processor
address request
virtual address
0
b
not currently in physical memory
physical address
0
b′

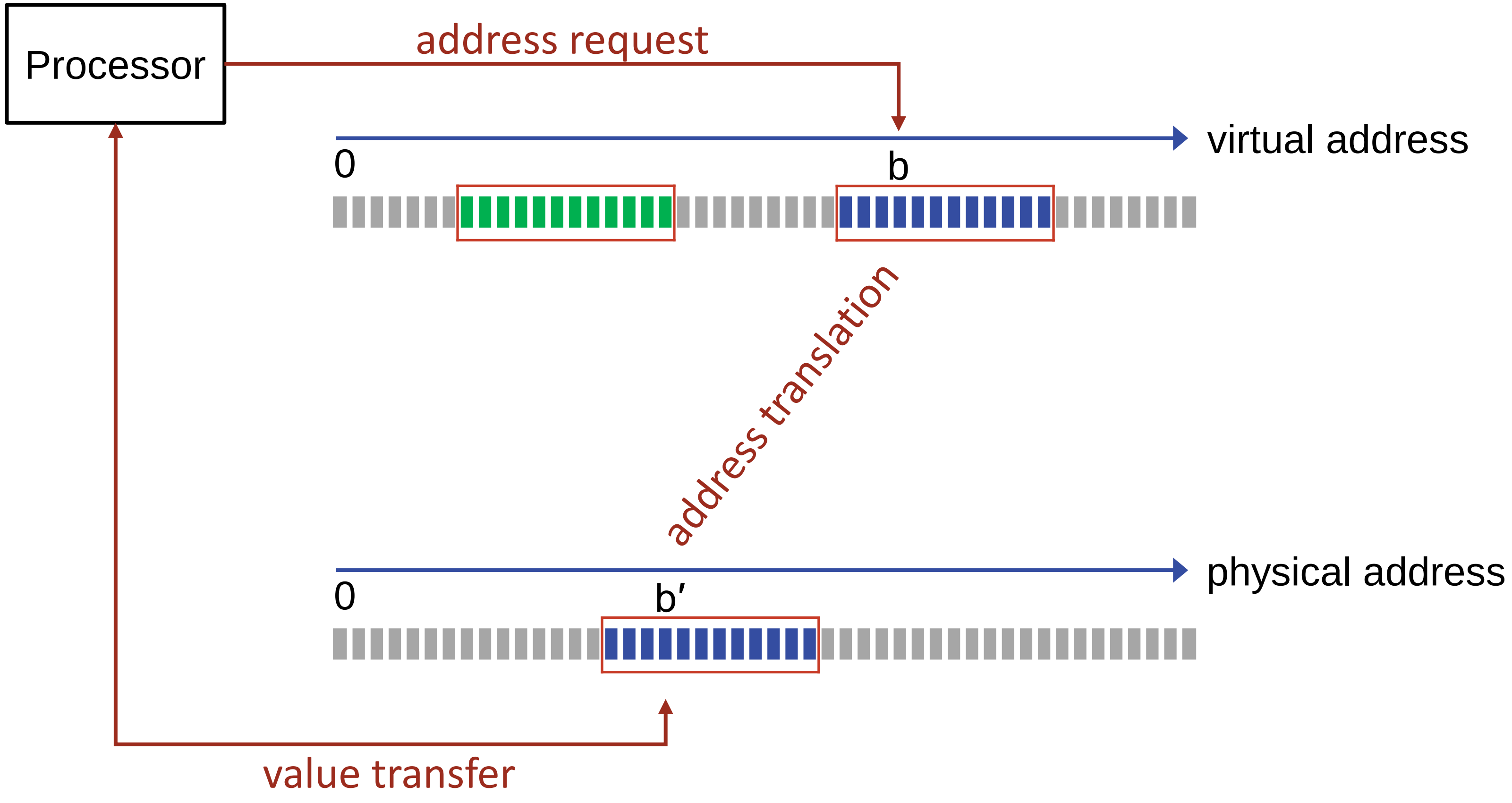
Concepts – Hardware & OS




Processor
address request
virtual address
0
b
page in
physical address
0
b′

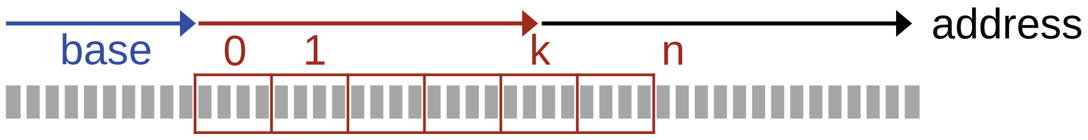
Concepts – Hardware & OS




Processor
address request
virtual address
0
b
address translation
physical address
0
b′
value transfer

cache memory. A cache is a small but very fast memory that temporarily represents bytes of physical memory. Abstractly, address mapping from physical to cache memory is similar to address mapping from virtual memory to physical memory.

array layout. One-dimensional arrays of length *n* of *m*-byte elements are typically laid out in *n* consecutive *m*-byte groups, starting at some base address in virtual memory:



Access to the *k*th array element requires computing its virtual address as *base+m·k*, and then using an operation that either loads or stores values at that location, where the corresponding physical-memory location is obtained by address mapping.

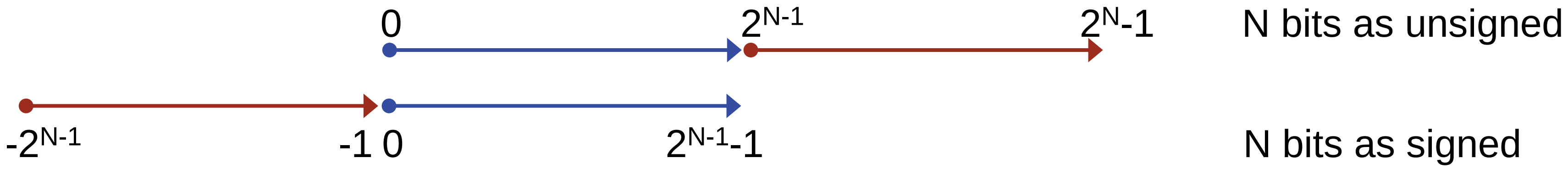
The simple model whereby the time to access an array element A[*k*] is constant, and is independent of the value of *k*, assumes that entire array already resides in physical memory, and ignores the time involved in paging regions of the array into physical memory.

numerical representation. A numerical representation is a convention whereby a sequence of bits is interpreted as the representation of a number. There are two principal forms of numerical representation: fixed point and floating point. Each form has two varieties: 32-bit and 64-bit.

fixed-point binary integer. A fixed-point binary integer is a sequence of bits interpreted positionally as powers of 2. Thus, just as the decimal fixed-point integer 101 represents $(1 \cdot 10^2)+(0 \cdot 10^1)+(1 \cdot 10^0)$, i.e., a hundred and one, so the binary fixed-point integer 101 represents $(1 \cdot 2^2)+(0 \cdot 2^1)+(1 \cdot 2^0)$, i.e., five.

When N-bits are interpreted as an unsigned integer, they can represent 0 through 2^{N-1} The types **`int`** and **`long`** are 32-bit and 64-bit two's-complement fixed-point integers, respectively.

two's-complement integer. A convention for the representation of signed integers in N bits. A leading 0 bit followed by the remaining N-1 bits is interpreted as a positive (N-1)-bit binary integer, and a leading 1 bit followed by the remaining N-1 bits is interpreted as a negative binary integer.



In this case, instead of the next number after $2^{N-1}-1$ being interpreted as the positive number, 2^{N-1}, it is interpreted as the most negative negative number, -2^{N-1}. Continuing "up", we eventually reach all 1s, which as an unsigned integer would be the largest value, but in two's complement, is interpreted as -1.

floating-point number. A floating-point number is a number in scientific notation. It consists of a signed mantissa, and a signed exponent. In base-2, if the value of the mantissa is m, and the value of the exponent is e, then the number represented is $m \cdot 2^e$.

The **`float`** type has 32 bits, and the **`double`** type has 64 bits. The exact interpretation of bits need not be understood. Suffice it to say that **`double`** has more bits than **`float`** for both mantissa and exponent.

The correspondence between binary (base-2) floating-point numbers (used internally) and decimal (base-10) floating-point numbers (used externally for input and output) is approximate.

character set. A character set is an encoding of symbols as a sequence of bits, e.g., Unicode.

Unicode. The international Unicode standard is a character set intended to represent almost every known symbol on Earth, including many emojis. The most common 65,536 symbols are representable as a **char**, and are encoded in two bytes. The remaining symbols are encoded as two **char** values.