12 Operating System

Civilization progresses by extending the number of operations that we can perform without thinking about them.
—Alfred North Whitehead, *Introduction to Mathematics* (1911)

In chapters 1–6 we described and built a general-purpose hardware architecture. In chapters 7–11 we developed a software hierarchy that makes the hardware usable, culminating in the construction of a modern, object-based language. Other high-level programming languages can be specified and implemented on top of the hardware platform, each requiring its own compiler.

The last major piece missing in this puzzle is an *operating system*. The OS is designed to close gaps between the computer's hardware and software, making the computer system more accessible to programmers, compilers, and users. For example, to display the text Hello World on the screen, several hundred pixels must be drawn at specific screen locations. This can be done by consulting the hardware specification and writing code that turns bits on and off in selected RAM locations. Clearly, high-level programmers expect a better interface. They want to write print ("Hello World") and let someone else worry about the details. That's where the operating system enters the picture.

Throughout this chapter, the term *operating system* is used rather loosely. Our OS is minimal, aiming at (i) encapsulating low-level hardware-specific services in high-level programmer-friendly software services and (ii) extending high-level languages with commonly used functions and abstract data types. The dividing line between an *operating system* in this sense and a *standard class library* is not clear. Indeed, modern programming languages pack many standard operating system services like graphics, memory management, multitasking, and numerous other extensions into what is known as the language's *standard class library*. Following this model, the Jack OS is packaged as a collection of supporting classes, each providing a set of related services via Jack subroutine calls. The complete OS API is given in appendix 6.

High-level programmers expect the OS to deliver its services through well-designed interfaces that hide the gory hardware details from their application programs. To do so, the OS code must operate close to the hardware, manipulating memory, input/output, and processing devices almost directly. Further, because the OS supports the execution of every program that runs on the computer, it must be highly efficient. For example, application

programs create and dispose objects and arrays all the time. Therefore, we better do it quickly and economically. Any gain in the time- and space-efficiency of an enabling OS service can impact dramatically the performance of all the application programs that depend on it.

Operating systems are usually written in a high-level language and compiled into binary form. Our OS is no exception—it is written in Jack, just like Unix was written in C. Like the C language, Jack was designed with sufficient "lowness" in it, permitting an intimate closeness to the hardware when needed.

The chapter starts with a relatively long Background section that presents key algorithms normally used is OS implementations. These include mathematical operations, string manipulations, memory management, text and graphics output, and keyboard input. This algorithmic introduction is followed by a Specification section describing the Jack OS, and an Implementation section that offers guidance on how to build the OS using the algorithms presented. As usual, the Project section provides the necessary guidelines and materials for gradually constructing and unit-testing the entire OS.

The chapter embeds key lessons in system-oriented software engineering and in computer science. On the one hand, we describe programming techniques for developing low-level system services, as well as "programming at the large" techniques for integrating and streamlining the OS services. On the other hand, we present a set of elegant and highly efficient algorithms, each being a computer science gem.

12.1 Background

Computers are typically connected to a variety of input/output devices such as a keyboard, screen, mouse, mass storage, network interface card, microphone, speakers, and more. Each of these I/O devices has its own electromechanical idiosyncrasies; thus, reading and writing data on them involves many technical details. High-level languages abstract away these details by offering high-level abstractions like `let n = Keyboard.readInt("Enter a number:")`. Let's delve into what should be done in order to realize this seemingly simple data-entry operation.

First, we engage the user by displaying the prompt `Enter a number:`. This entails creating a `String` object and initializing it to the array of `char` values `'E'`, `'n'`, `'t'`, …, and so on. Next, we have to render this string on the screen, one character at a time, while updating the *cursor* position for keeping track of where the next character should be physically displayed. After displaying the `Enter a number:` prompt, we have to stage a loop that waits until the user will oblige to press some keys on the keyboard—hopefully keys that represent digits. This requires knowing how to (i) capture a keystroke, (ii) get a single character input, (iii) append these characters to a string, and (iv) convert the string into an integer value.

If what has been elaborated so far sounds arduous, the reader should know that we were actually quite gentle, sweeping many gory details under the rug. For example, what exactly is meant by "creating a string object," "displaying a character on the screen," and "getting a multicharacter input"?

Let's start with "creating a string object." String objects don't pop out of thin air, fully formed. Each time we want to create an object, we must find available space for representing the object in the RAM, mark this space as used, and remember to free it when the object is no longer needed. Proceeding to the "display a character" abstraction, note that characters cannot be displayed. The only things that can be physically displayed are individual pixels. Thus, we have to figure out what is the character's *font*, compute where the bits that represent the font image can be found in the screen memory map, and then turn these bits on and off as needed. Finally, to "get a multicharacter input," we have to enter a loop that not only listens to the keyboard and accumulates characters as they come along but also allows the user to backspace, delete, and retype characters, not to mention the need to echo each of these actions on the screen for visual feedback.

The agent that takes care of this elaborate behind-the-scenes work is the operating system. The execution of the statement `let n = Keyboard.readInt("Enter a number:")` entails many OS function calls, dealing with diverse issues like memory allocation, input driving, output driving, and string processing. Compilers use the OS services abstractly by injecting OS function calls into the compiled code, as we saw in the previous chapter. In this chapter we explore *how these functions are actually realized*. Of course, what we have surveyed so far is just a small subset of the OS responsibilities. For example, we didn't mention mathematical operations, graphical output, and other commonly needed services. The good news is that a well-written OS can integrate these diverse and seemingly unrelated tasks in an elegant and efficient way, using cool algorithms and data structures. That's what this chapter is all about.

12.1.1 Mathematical Operations

The four arithmetic operations *addition*, *subtraction*, *multiplication*, and *division* lie at the core of almost every computer program. If a loop that executes a million times contains expressions that use some of these operations, they'd better be implemented efficiently.

Normally, addition is implemented in hardware, at the ALU level, and subtraction is gained freely, courtesy of the two's complement method. Other arithmetic operations can be handled either by hardware or by software, depending on cost/performance considerations. We now turn to present efficient algorithms for computing multiplication, division, and square roots. These algorithms lend themselves to both software and hardware implementations.

Efficiency First

Mathematical algorithms are made to operate on n-bit values, n typically being 16, 32, or 64 bits, depending on the operands' data types. As a rule, we seek algorithms whose running time is a polynomial function of this word size n. Algorithms whose running time depends on the *values* of n-bit numbers are unacceptable, since these values are exponential in n. For example, suppose we implement the multiplication operation $x \times y$ naïvely, using the

repeated addition algorithm `for i = 1 … y {sum = sum + x}`. If y is 64-bit wide, its value may well be greater than 9,000,000,000,000,000,000, implying that the loop may run for billions of years before terminating.

In sharp contrast, the running times of the multiplication, division, and square root algorithms that we present below depend not to the n-bit *values* on which they are called to operate, which may be as large as 2^n, but rather on n, the number of their bits. When it comes to efficiency of arithmetic operations, that's the best that we can possibly hope for.

We will use the *Big-O* notation, $O(n)$, to describe a running time which is "in the order of magnitude of n." The running time of all the arithmetic algorithms that we present in this chapter is $O(n)$, where n is the bit width of the inputs.

Multiplication

Consider the standard multiplication method taught in elementary school. To compute 356 times 73, we line up the two numbers one on top of the other, right-justified. Next, we multiply 356 by 3. Next, we shift 356 to the left one position, and multiply 3560 by 7 (which is the same as multiplying 356 by 70). Finally, we sum up the columns and obtain the result. This procedure is based on the insight that $356 \times 73 = 356 \times 70 + 356 \times 3$. The binary version of this procedure is illustrated in figure 12.1, using another example.

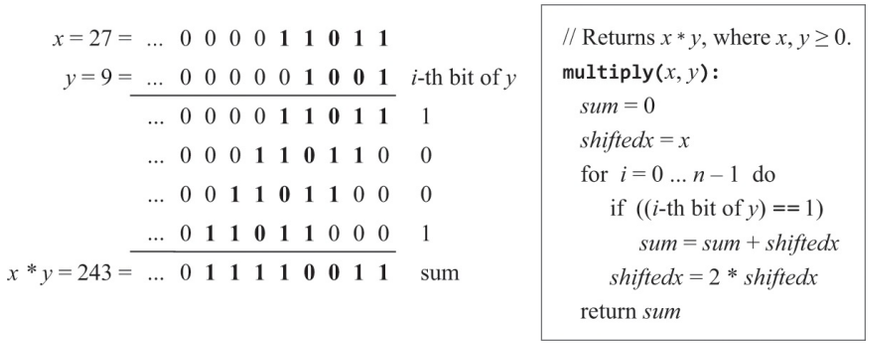


Figure 12.1 Multiplication algorithm.

Notation note: The algorithms presented in this chapter are written in a self-explanatory pseudocode syntax. We use indentation to mark blocks of code, obviating the need for curly brackets or begin/end keywords. For example, in figure 12.1, $sum = sum + shiftedx$ belongs to the single-statement body of the `if` logic, and $shiftedx = 2 * shiftedx$ ends the two-statement body of the `for` logic.

Let's inspect the multiplication procedure illustrated at the left of figure 12.1. For each i-th bit of y, we shift x i times to the left (same as multiplying x by 2^i). Next, we look at the i-th bit of y: If it is 1, we add the shifted x to an accumulator; otherwise, we do nothing. The algorithm shown on the right formalizes this procedure. Note that 2 * *shiftedx* can be computed efficiently either by left-shifting the bitwise representation of *shiftedx* or by adding

shiftedx to itself. Either operation lends itself to primitive hardware operations.

Running time: The multiplication algorithm performs *n* iterations, where *n* is the bit width of the *y* input. In each iteration, the algorithm performs a few addition and comparison operations. It follows that the total running time of the algorithm is $a + b \cdot n$, where *a* is the time it takes to initialize a few variables, and *b* is the time it takes to perform a few addition and comparison operations. Formally, the algorithm's running time is *O* (*n*), where *n* is the bit width of the inputs.

To reiterate, the running time of this $x \times y$ algorithm does not depend on the *values* of the *x* and *y* inputs; rather, it depends on the *bit width* of the inputs. In computers, the bit width is normally a small fixed constant like 16 (short), 32 (int), or 64 (long), depending on the data types of the inputs. In the Hack platform, the bit width of all data types is 16. If we assume that each iteration of the multiplication algorithm entails about ten Hack machine instructions, it follows that each multiplication operation will require at most 160 clock cycles, irrespective of the size of the inputs. In contrast, algorithms whose running time is proportional not to the bit width but rather to the values of the inputs will require $10 \cdot 2^{16} = 655{,}360$ clock cycles.

Division

The naïve way to compute the division of two *n*-bit numbers *x* / *y* is to count how many times *y* can be subtracted from *x* until the remainder becomes less than *y*. The running time of this algorithm is proportional to the value of the dividend *x* and thus is unacceptably exponential in the number of bits *n*.

To speed things up, we can try to subtract large chunks of *y*'s from *x* in each iteration. For example, suppose we have to divide 175 by 3. We start by asking: What is the largest number, $x = (90, 80, 70, \ldots, 20, 10)$, so that $3 \cdot x \leq 175$? The answer is 50. In other words, we managed to subtract fifty 3's from 175, shaving fifty iterations from the naïve approach. This accelerated subtraction leaves a remainder of $175 - 3 \cdot 50 = 25$. Moving along, we now ask: What is the largest number, $x = (9, 8, 7, \ldots 2, 1)$, so that $3 \cdot x \leq 25$? The answer is 8, so we managed to make eight additional subtractions of 3, and the answer, so far, is $50 + 8 = 58$.The remainder is $25 - 3 \cdot 8 = 1$, which is less than 3, so we stop the process and announce that $175/3 = 58$ with a remainder of 1.

This technique is the rationale behind the dreaded school procedure known as *long division*. The binary version of this algorithm is identical, except that instead of accelerating the subtraction using powers of 10 we use powers of 2. The algorithm performs *n* iterations, *n* being the number of digits in the dividend, and each iteration entails a few multiplication (actually, shifting), comparison, and subtraction operations. Once again, we have an x/y algorithm whose running time does not depend on the values of *x* and *y*. Rather, the running time is *O* (*n*), where *n* is the bit width of the inputs.

Writing down this algorithm as we have done for multiplication is an easy exercise. To make things interesting, figure 12.2 presents another division algorithm which is as efficient, but more elegant and easier to implement.

```
// Returns the integer division x / y,
// where x ≥ 0 and y > 0.
divide(x, y):
    if (y > x) return 0
    q = divide (x, 2 * y)
    if ((x – 2 * q * y) < y)
        return 2 * q
    else
        return 2 * q + 1
```

Figure 12.2 Division algorithm.

Suppose we have to divide 480 by 17. The algorithm shown in figure 12.2 is based on the insight $480/17 = 2\cdot(240/17) = 2\cdot(2\cdot(120/17)) = 2\cdot(2\cdot(2\cdot(60/17))) = \cdots$, and so on. The depth of this recursion is bounded by the number of times y can be multiplied by 2 before reaching x. This also happens to be, at most, the number of bits required to represent x. Thus, the running time of this algorithm is $O(n)$, where n is the bit width of the inputs.

One snag in this algorithm is that each multiplication operation also requires $O(n)$ operations. However, an inspection of the algorithm's logic reveals that the value of the expression $(2 * q * y)$ can be computed without multiplication. Instead, it can be obtained from its value in the previous recursion level, using addition.

Square Root

Square roots can be computed efficiently in a number of different ways, for example, using

the Newton-Raphson method or a Taylor series expansion. For our purpose, though, a simpler algorithm will suffice. The square root function $y = \sqrt{x}$ has two attractive properties. First, it is monotonically increasing. Second, its inverse function, $y = x^2$, is a function that we already know how to compute efficiently—multiplication. Taken together, these properties imply that we have all we need to compute square roots efficiently, using a form of *binary search*. Figure 12.3 gives the details.

```
// Computes the integer part of y = √x
// Strategy: finds an integer y such that y^2 ≤ x < (y+1)^2  (for 0 ≤ x < 2^n)
// by performing binary search in the range 0 ... 2^(n/2) – 1
sqrt(x):
   y = 0
   for  j = (n/2 – 1) ... 0  do
      if (y+2^j)^2 ≤ x then y = y+2^j
   return y
```

Figure 12.3 Square root algorithm.

Since the number of iterations in the binary search that the algorithm performs is bound by $n / 2$ where n is the number of bits in x, the algorithm's running time is $O(n)$.

To sum up this section about mathematical operations, we presented algorithms for computing multiplication, division, and square root. The running time of each of the algorithms is $O(n)$, where n is the bit width of the inputs. We also observed that in computers, n is a small constant like 16, 32, or 64. Therefore, every addition, subtraction, multiplication, and division operation can be carried out swiftly, in a predictable time that is unaffected by the magnitude of the inputs.

12.1.2 Strings

In addition to primitive data types, most programming languages feature a *string* type designed to represent sequences of characters like "Loading game …" and "QUIT". Typically, the string abstraction is supplied by a String class that is part of the standard class library that supports the language. This is also the approach taken by Jack.

All the string constants that appear in Jack programs are implemented as String objects. The String class, whose API is documented in appendix 6, features various string processing methods like appending a character to the string, deleting the last character, and so on. These services are not difficult to implement, as we'll describe later in the chapter. The more challenging String methods are those that convert integer values to strings and strings of digit characters to integer values. We now turn to discuss algorithms that carry out these operations.

String representation of numbers: Computers represent numbers internally using binary codes. Yet humans are used to dealing with numbers that are written in decimal notation. Thus, when humans have to read or input numbers, *and only then*, a conversion to or from decimal notation must be performed. When such numbers are captured from an input device like a keyboard, or rendered on an output device like a screen, they are cast as strings of characters, each representing one of the digits 0 to 9. The subset of relevant characters is:

Character:	'0'	'1'	'2'	'3'	'4'	'5'	'6'	'7'	'8'	'9'
Character code:	48	49	50	51	52	53	54	55	56	57

(The complete Hack character set is given in appendix 5). We see that digit characters can be easily converted into the integers that they represent, and vice versa. The integer value of character c, where $48 \le c \le 57$, is $c - 48$. Conversely, the character code of the integer x, where $0 \le x \le 9$, is $x + 48$.

Once we know how to handle single-digit characters, we can develop algorithms for converting any integer into a string and any string of digit characters into the corresponding integer. These conversion algorithms can be based on either iterative or recursive logic, so figure 12.4 presents one of each.

int to string:

```
// Returns the string representation of
// a nonnegative integer.
int2String(val):
   lastDigit = val % 10
   c = character representing lastDigit
   if (val < 10)
     return  c  (as a string)
   else
     return  int2String (val / 10).appendChar(c)
```

string to int:

```
// Returns the integer value of a string
// of digit characters, assuming that str[0]
// represents the most significant digit.
string2Int(str):
    val = 0
    for ( i = 0 ... str.length() ) do
        d = integer value of str.charAt(i)
        val = val * 10 + d
    return  val
```

Figure 12.4 String-integer conversions. (appendChar, length, and charAt are String class methods.)

It is easy to infer from figure 12.4 that the running times of the int2String and string2Int algorithms are $O(n)$, where n is the number of the digit-characters in the input.

12.1.3 Memory Management

Each time a program creates a new array or a new object, a memory block of a certain size must be allocated for representing the new array or object. And when the array or object is no longer needed, its RAM space may be recycled. These chores are done by two classical OS functions called alloc and deAlloc. These functions are used by compilers when generating low-level code for handling constructors and destructors, as well as by high-level programmers, as needed.

The memory blocks for representing arrays and objects are carved from, and recycled back into, a designated RAM area called a *heap*. The agent responsible for managing this resource is the operating system. When the OS starts running, it initializes a pointer named `heapBase`, containing the heap's base address in the RAM (in Jack, the heap starts just after the stack's end, with `heapBase=2048`). We'll present two heap management algorithms: basic and improved.

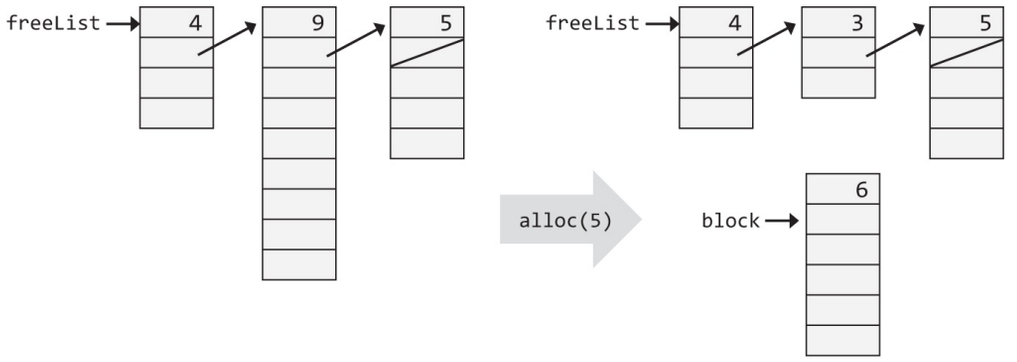
Memory allocation algorithm (basic): The data structure that this algorithm manages is a single pointer, named `free`, which points to the beginning of the heap segment that was not yet allocated. See figure 12.5a for the details.

```
init():
    free = heapBase

// Allocates a memory block of size words.
alloc(size):
    block = free
    free = free + size
    return block

// Frees the memory space of the given object.
deAlloc(object):
    do nothing
```

Figure 12.5a Memory allocation algorithm (basic).

The basic heap management scheme is clearly wasteful, as it never reclaims any memory space. But, if your application programs use only a few small objects and arrays, and not too many strings, you may get away with it.

Memory allocation algorithm (improved): This algorithm manages a linked list of available memory segments, called `freeList` (see figure 12.5b). Each segment in the list begins with two housekeeping fields: the segment's *length* and a pointer to the *next* segment in the list.




```
init():
   freeList = heapBase
   freeList.size = heapSize
   freeList.next = 0

// Allocates a memory block of size words.
alloc(size):
   search freeList using best-fit or first-fit heuristics
        to obtain a segment with segment.size ≥ size + 2
   if no such segment is found, return failure
        (or attempt defragmentation)
   block = base address of the found space
   update the freeList and the fields of block
        to account for the allocation
   return block

// Frees the memory space of the given object.
deAlloc(object):
   append object to the end of the freeList
```

Figure 12.5b Memory allocation algorithm (improved).

When asked to allocate a memory block of a given size, the algorithm has to search the freeList for a suitable segment. There are two heuristics for doing this search. *Best-fit* finds the shortest segment that is long enough for representing the required size, while *first-fit* finds the first segment that is long enough. Once a suitable segment has been found, the required memory block is carved from it (the location just before the beginning of the returned block, block[−1], is reserved to hold its length, to be used during deallocation).

Next, the *length* of this segment is updated in the freeList, reflecting the length of the part that remained after the allocation. If no memory was left in the segment, or if the remaining part is practically too small, the entire segment is eliminated from the freeList.

When asked to reclaim the memory block of an unused object, the algorithm appends the deallocated block to the end of the freeList.

Dynamic memory allocation algorithms like the one shown in figure 12.5b may create block fragmentation problems. Hence, a *defragmentation* operation should be considered, that is, merging memory areas that are physically adjacent in memory but logically split into different segments in the freeList. The defragmentation can be done each time an object is deallocated, when alloc() fails to find a block of the requested size, or according to some other,

periodical ad hoc condition.

Peek and poke: We end the discussion of memory management with two simple OS functions that have nothing to do with resource allocation. Memory.peek(*addr*) returns the value of the RAM at address *addr*, and Memory.poke(*addr*,*value*) sets the word in RAM address *addr* to *value*. These functions play a role in various OS services that manipulate the memory, including graphics routines, as we now turn to discuss.

12.1.4 Graphical Output

Modern computers render graphical output like animation and video on high-resolution color screens, using optimized graphics drivers and dedicated graphical processing units (GPUs). In Nand to Tetris we abstract away most of this complexity, focusing instead on fundamental graphics-drawing algorithms and techniques.

We assume that the computer is connected to a physical black-and-white screen arranged as a grid of rows and columns, and at the intersection of each lies a pixel. By convention, the columns are numbered from left to right and the rows are numbered from top to bottom. Thus pixel (0,0) is located at the screen's top-left corner.

We assume that the screen is connected to the computer system through a *memory map*—a dedicated RAM area in which each pixel is represented by one bit. The screen is refreshed from this memory map many times per second by a process that is external to the computer. Programs that simulate the computer's operations are expected to emulate this refresh process.

The most basic operation that can be performed on the screen is drawing an individual pixel specified by (*x*,*y*) coordinates. This is done by turning the corresponding bit in the memory map on or off. Other operations like drawing a line and drawing a circle are built on top of this basic operation. The graphics package maintains a *current color* that can be set to *black* or *white*. All the drawing operations use the current color.

Pixel drawing (drawPixel): Drawing a selected pixel in screen location (*x*,*y*) is achieved by locating the corresponding bit in the memory map and setting it to the current color. Since the RAM is an *n*-bit device, this operation requires reading and writing an *n*-bit value. See figure 12.6.

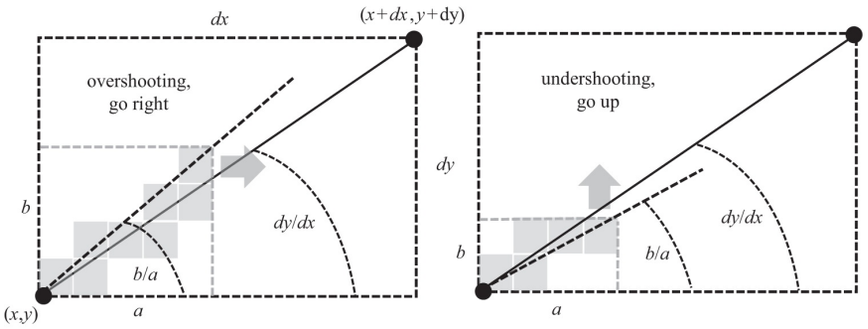
```
// Sets pixel (x,y) to the current color.
drawPixel(x,y):
    Using x and y, compute the RAM address where
        the pixel is represented;
    Using Memory.peek, get the 16-bit value of
        this address;
    Using some bitwise operation, set (only) the bit
        that corresponds to the pixel to the current color;
    Using Memory.poke, write the modified 16-bit
        value "back" to the RAM address.
```

Figure 12.6 Drawing a pixel.

The memory map interface of the Hack screen is specified in section 5.2.4. This mapping should be used in order to realize the drawPixel algorithm.

Line drawing (drawLine): When asked to render a continuous "line" between two "points" on a grid made of discrete pixels, the best that we can possibly do is approximate the line by drawing a series of pixels along the imaginary line connecting the two points. The "pen" that we use for drawing the line can move in four directions only: up, down, left, and right. Thus, the drawn line is bound to be jagged, and the only way to make it look good is to use a high-resolution screen with the tiniest possible pixels. Note, though, that the human eye, being yet another machine, also has a limited image-capturing capacity, determined by the number and type of receptor cells in the retina. Thus, high-resolution screens can fool the human brain to believe that the lines made of pixels are visibly smooth. In fact they are always jagged.

The procedure for drawing a line from $(x1,y1)$ to $(x2,y2)$ starts by drawing the $(x1,y1)$ pixel and then zigzagging in the direction of $(x2,y2)$ until that pixel is reached. See figure 12.7.




```
// Draws a line between two given points.
drawLine(x1, y1, x2, y2):
    set x to x1, y to y1, compute dx and dy.
    // a and b track how many times we
    // went right, and up, so far.
    a = 0, b = 0
    while ((a ≤ dx) and (b ≤ dy))
        drawPixel(x + a, y + b)
        // Goes right, or up:
        if (b/a > dy/dx)   { a=a+1 }
        else               { b=b+1 }
```

```
// Draws a line between two points (improved).
drawLine(x1, y1, x2, y2):
    set x to x1, y to y1, compute dx and dy.
    // a and b track how many times we
    // went right, and up, so far.
    a = 0, b = 0
    diff = 0
    while ((a ≤ dx) and (b ≤ dy))
        drawPixel(x + a, y + b)
        // Goes right, or up:
        if (diff < 0)  { a=a+1, diff = diff + dy }
        else           { b=b+1, diff = diff – dx }
```

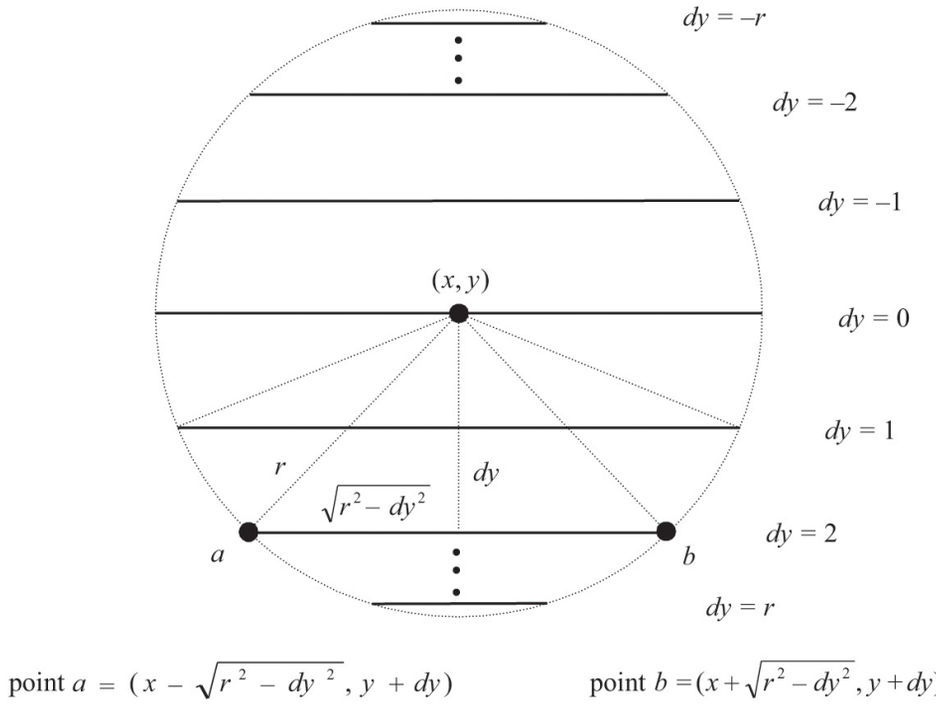
involves only addition and subtraction operations

Figure 12.7 Line-drawing algorithm: basic version (bottom, left) and improved version (bottom, right).

The use of two division operations in each loop iteration makes this algorithm neither efficient nor accurate. The first obvious improvement is replacing the $b / a > dy / dx$ condition with the equivalent $a \cdot dy < b \cdot dx$, which requires only integer multiplication. Careful inspection of the latter condition reveals that it may be checked without any multiplication. As shown in the improved algorithm in figure 12.7, this may be done efficiently by maintaining a variable that updates the value of $(a \cdot dy - b \cdot dx)$ each time a or b is incremented.

The running time of this line-drawing algorithm is $O(n)$, where n is the number of pixels along the drawn line. The algorithm uses only addition and subtraction operations and can be implemented efficiently in either software or hardware.

Circle drawing (`drawCircle`): Figure 12.8 presents an algorithm that uses three routines that we've already implemented: multiplication, square root, and line drawing.




```
// Draws a filled circle of radius r, centered at (x,y).
drawCircle(x, y, r):
    for each dy = -r to r do:
        drawLine((x – √(r² – dy²), y + dy), (x + √(r² – dy²), y + dy))
```

Figure 12.8 Circle-drawing algorithm.

The algorithm is based on drawing a sequence of horizontal lines (like the typical line *ab* in the figure), one for each row in the range $y-r$ to $y+r$. Since r is specified in pixels, the algorithm ends up drawing a line in every row along the circle's north-south diameter, resulting in a completely filled circle. A simple tweak can cause this algorithm to draw only the circle's outline, if so desired.

12.1.5 Character Output

To develop a capability for displaying characters, we first turn our physical, pixel-oriented screen into a logical, character-oriented screen suitable for rendering fixed, bitmapped images that represent characters. For example, consider a physical screen that features 256 rows of 512 pixels each. If we allocate a grid of 11 rows by 8 columns for drawing a single character, then our screen can display 23 lines of 64 characters each, with 3 extra rows of pixels left unused.

Fonts: The character sets that computers use are divided into *printable* and *non-printable*

subsets. For each printable character in the Hack character set (see appendix 5), an 11-row-by-8-column bitmap image was designed, to the best of our limited artistic abilities. Taken together, these images are called a *font*. Figure 12.9 shows how our font renders the uppercase letter N. To handle character spacing, each character image includes at least a 1-pixel space before the next character in the row and at least a 1-pixel space between adjacent rows (the exact spacing varies with the size and squiggles of individual characters). The Hack font consists of ninety-five such bitmap images, one for each printable character in the Hack character set.

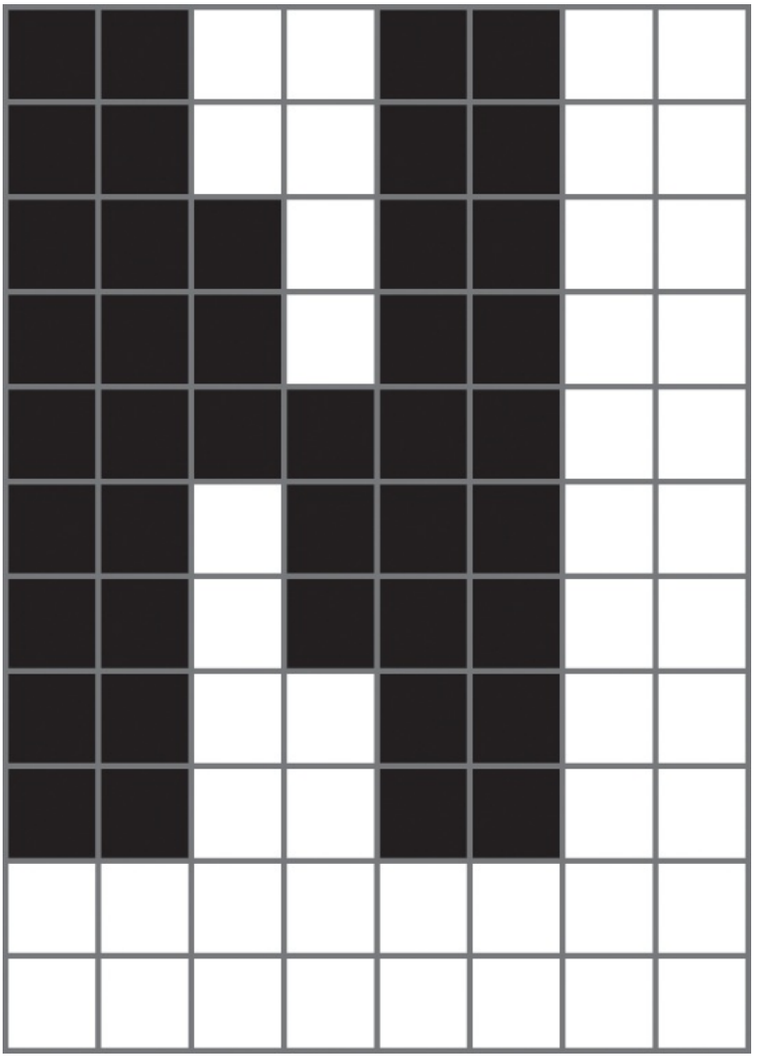


Figure 12.9 Example of a character bitmap.

Font design is an ancient yet vibrant art. The oldest fonts are as old as the art of writing, and new ones are routinely introduced by type designers who wish to make an artistic statement or solve a technical or functional objective. In our case, the small physical screen, on the one hand, and the wish to display a reasonable number of characters in each row, on the other, led to the pragmatic choice of a frugal image area of 11×8 pixels. This economy forced us to design a crude font, which nonetheless serves its purpose well.

Cursor: Characters are usually displayed one after the other, from left to right, until the end of the line is reached. For example, consider a program in which the statement print("a") is followed (perhaps not immediately) by the statement print ("b"). This implies that the program wants to display ab on the screen. To effect this continuity, the character-writing package maintains a global *cursor* that keeps track of the screen location where the next character should be drawn. The cursor information consists of column and row counts, say, cursor.col and cursor.row. After a character has been displayed, we do cursor.col++. At the end of the row we do cursor.row++ and cursor.col = 0. When the bottom of the screen is reached, there is a question of what to do next. Two possible actions are effecting a scrolling operation or clearing the screen and starting over by setting the cursor to (0,0).

To recap, we described a scheme for writing individual characters on the screen. Writing other types of data follows naturally from this basic capability: strings are written character by character, and numbers are first converted to strings and then written as strings.

12.1.6 Keyboard Input

Capturing inputs that come from the keyboard is more intricate than meets the eye. For example, consider the statement let name = Keyboard.readLine("enter your name:"). By definition, the execution of the readLine function depends on the dexterity and collaboration of an unpredictable entity: a human user. The function will not terminate until the user has pressed some keys on the keyboard, ending with an ENTER. The problem is that humans press and release keyboard keys for variable and unpredictable durations of time, and often take a coffee break in the middle. Also, humans are fond of backspacing, deleting, and retyping characters. The implementation of the readLine function must handle all these irregularities.

This section describes how keyboard input is managed, in three levels of abstraction: (i) detecting which key is currently pressed on the keyboard, (ii) capturing a single-character input, and (iii) capturing a multicharacter input.

Detecting keyboard input (keyPressed): Detecting which key is presently pressed is a hardware-specific operation that depends on the keyboard interface. In the Hack computer, the keyboard continuously refreshes a 16-bit memory register whose address is kept in a pointer named KBD. The interaction contract is as follows: If a key is currently pressed on the keyboard, that address contains the key's character code (the Hack character set is given in appendix 5); otherwise, it contains 0. This contract is used for implementing the keyPressed function shown in figure 12.10.

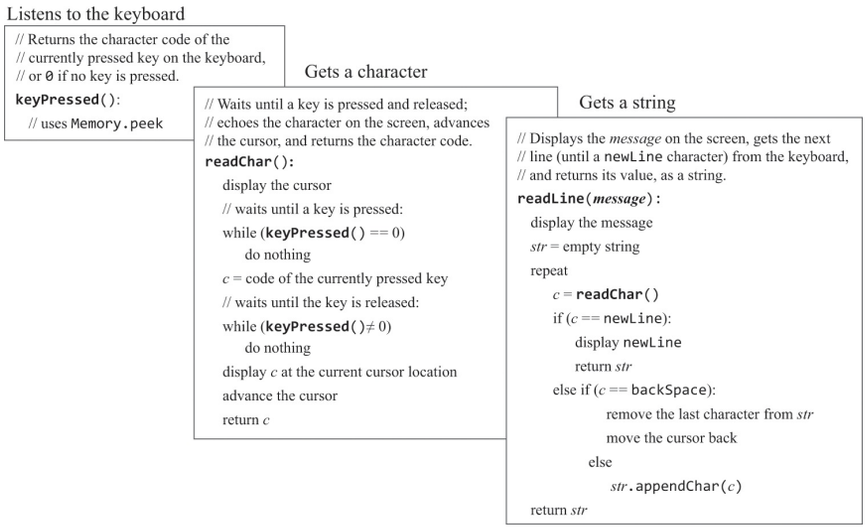


Figure 12.10 Handling input from the keyboard.

Reading a single character (readChar): The elapsed time between the *key pressed* and the subsequent *key released* events is unpredictable. Hence, we have to write code that neutralizes this uncertainty. Also, when users press keys on the keyboard, we want to give feedback as to which keys have been pressed (something that you have probably grown to take for granted). Typically, we want to display some graphical cursor at the screen location where the next input goes, and, after some key has been pressed, we want to echo the inputted character by displaying its bitmap on the screen at the cursor location. All these actions are implemented by the readChar function.

Reading a string (readLine): A multicharacter input typed by the user is considered final after the ENTER key has been pressed, yielding the newLine character. Until the ENTER key is pressed, the user should be allowed to backspace, delete, and retype previously typed characters. All these actions are accommodated by the readLine function.

As usual, our input-handling solutions are based on a cascading series of abstractions: The high-level program relies on the readLine abstraction, which relies on the readChar abstraction, which relies on the keyPressed abstraction, which relies on the Memory.peek abstraction, which relies on the hardware.

12.2 The Jack OS Specification

The previous section presented algorithms that address various classical operating system tasks. In this section we turn to formally specify one particular operating system—the Jack OS. The Jack operating system is organized in eight classes:

- Math: provides mathematical operations
- String: implements the String type
- Array: implements the Array type

- Memory: handles memory operations
- Screen: handles graphics output to the screen
- Output: handles character output to the screen
- Keyboard: handles input from the keyboard
- Sys: provides execution-related services

The complete OS API is given in appendix 6. This API can be viewed as the OS specification. The next section describes how this API can be implemented using the algorithms presented in the previous section.

12.3 Implementation

Each OS class is a collection of subroutines (constructors, functions, and methods). Most of the OS subroutines are simple to implement and are not discussed here. The remaining OS subroutines are based on the algorithms presented in section 12.2. The implementation of these subroutines can benefit from some tips and guidelines, which we now turn to present.

Init functions: Some OS classes use data structures that support the implementation of some of their subroutines. For each such *OSClass*, these data structures can be declared statically at the class level and initialized by a fucntion which, by convention, we call *OSClass*.init. The init functions are for internal purposes and are not documented in the OS API.

Math

multiply: In each iteration *i* of the *multiplication* algorithm (see figure 12.1), the *i*-th bit of the second multiplicand is extracted. We suggest encapsulating this operation in a helper function bit (x,i) that returns true if the i-th bit of the integer x is 1, and false otherwise. The bit(x,i) function can be easily implemented using shifting operations. Alas, Jack does not support shifting. Instead, it may be convenient to define a fixed static array of length 16, say twoToThe, and set each element *i* to 2 raised to the power of *i*. The array can then be used to support the implementation of bit (x,i). The twoToThe array can be built by the Math.init function.

divide: The *multiplication* and *division* algorithms presented in figures 12.1 and 12.2 are designed to operate on nonnegative integers. Signed numbers can be handled by applying the algorithms to absolute values and setting the sign of the return values appropriately. For the multiplication algorithm, this is not needed: since the multiplicands are given in two's complement, their product will be correct with no further ado.

In the *division* algorithm, y is multiplied by a factor of 2, until $y > x$. Thus y can overflow. The overflow can be detected by checking when y becomes negative.

sqrt: In the *square root* algorithm (figure 12.3), the calculation of $(y + 2^j)^2$ can overflow,

resulting in an abnormally negative result. This problem can be addressed by changing efficiently the algorithm's if logic to: if $(y + 2^j)^2 \leq x$ and $(y + 2^j)^2 > 0$ then $y = y + 2^j$.

String

All the string constants that appear in a Jack program are realized as objects of the String class, whose API is documented in appendix 6. Specifically, each string is implemented as an object consisting of an array of char values, a maxLength property that holds the maximum length of the string, and a length property that holds the actual length of the string.

For example, consider the statement `let str="scooby"`. When the compiler handles this statement, it calls the String constructor, which creates a char array with `maxLength=6` and `Length=6`. If we later call the String method str.eraseLastChar(), the length of the array will become 5, and the string will effectively become "scoob". In general, then, array elements beyond length are not considered part of the string.

What should happen when an attempt is made to add a character to a string whose length equals maxLength? This issue is not defined by the OS specification: the String class may act gracefully and resize the array—or not; this is left to the discretion of individual OS implementations.

intValue, setInt: These subroutines can be implemented using the algorithms presented in figure 12.4. Note that neither algorithm handles negative numbers—a detail that must be handled by the implementation.

newLine, backSpace, doubleQuote: As seen in appendix 5, the codes of these characters are 128, 129, and 34.

The remaining String methods can be implemented straightforwardly by operating on the char array and on the length field that characterizes each String object.

Array

new: In spite of its name, this subroutine is not a constructor but rather a function. Therefore, the implementation of this function must allocate memory space for the new array by explicitly calling the OS function Memory.alloc.

dispose: This void method is called by statements like do arr.dispose(). The dispose implementation deallocates the array by calling the OS function Memory.deAlloc.

Memory

peek, poke: These functions provide direct access to the host memory. How can this low-level access be accomplished using the Jack high-level language? As it turns out, the Jack language includes a trapdoor that enables gaining complete control of the host computer's

memory. This trapdoor can be exploited for implementing Memory.peek and Memory.poke using plain Jack programming.

The trick is based on an anomalous use of a reference variable (pointer). Jack is a weakly typed language; among other quirks, it does not prevent the programmer from assigning a constant to a reference variable. This constant can then be treated as an absolute memory address. When the reference variable happens to be an array, this scheme provides indexed access to every word in the host RAM. See figure 12.11.

```
// Creates a Jack-level "proxy" of the RAM:
var Array memory;
let memory = 0;    // No problem . . .
...
// Gets the value of the RAM at address i:
let x = memory[i];
...
// Sets the value of the RAM at address i:
let memory[i] = 17;
...
```

Figure 12.11 A trapdoor enabling complete control of the host RAM from Jack.

Following the first two lines of code, the base of the memory array points to the first address in the computer's RAM (address 0). To get or set the value of the RAM location whose physical address is *i*, all we have to do is manipulate the array element memory[*i*]. This will cause the compiler to manipulate the RAM location whose address is $0 + i$, which is what we want.

Jack arrays are not allocated space on the heap at compile-time but rather at run-time, if and when the array's new function is called. Note that if new were a constructor and not a function, the compiler and the OS would have allocated the new array to some obscure address in the RAM that we cannot control. Like many classical hacks, this trick works because we use the array variable without initializing it properly, as is normally done when using arrays.

The memory array can be declared at the class level and initialized by the Memory.init function. Once this hack is done, the implementation of Memory.peek and Memory.poke becomes trivial.

alloc, deAlloc: These functions can be implemented by either one of the algorithms shown in

figures 12.5a and 12.5b. Either *best-fit* or *first-fit* can be used for implementing Memory.deAlloc.

The standard VM mapping on the Hack platform (see section 7.4.1) specifies that the *stack* be mapped on RAM addresses 256 to 2047. Thus the *heap* can start at address 2048.

In order to realize the freeList linked list, the Memory class can declare and maintain a static variable, freeList, as seen in figure 12.12. Although freeList is initialized to the value of heapBase (2048), it is possible that following several alloc and deAlloc operations freeList will become some other address in memory, as illustrated in the figure.

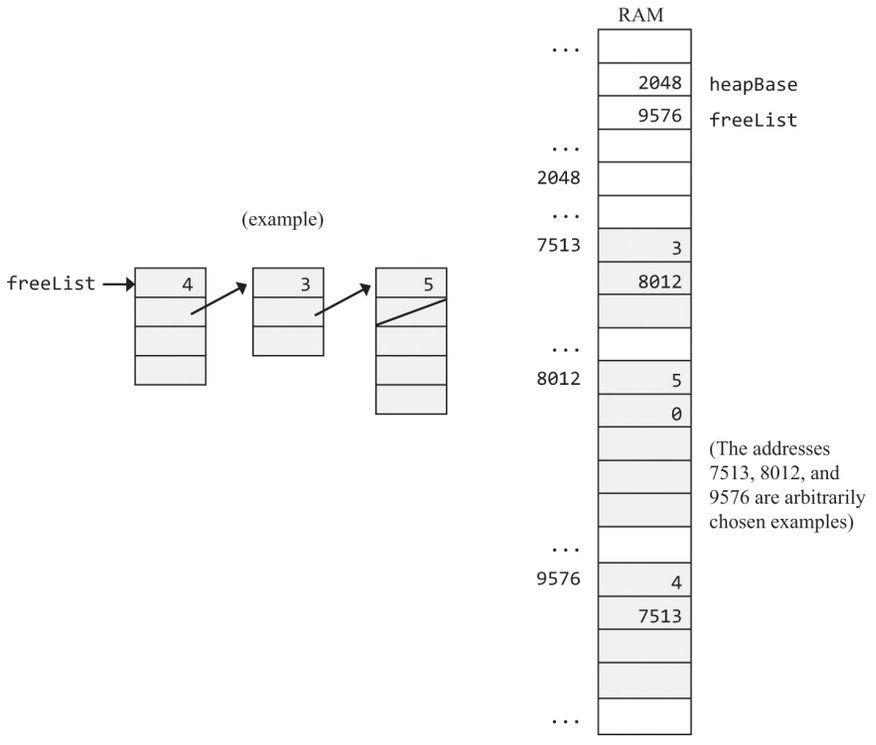


Figure 12.12 Logical view (left) and physical implementation (right) of a linked list that supports dynamic memory allocation.

For efficiency's sake, it is recommended to write Jack code that manages the freeList linked list directly in the RAM, as seen in figure 12.12. The linked list can be initialized by the Memory.init function.

Screen

The Screen class maintains a *current color* that is used by all the drawing functions of the class. The current color can be represented by a static Boolean variable.

drawPixel: Drawing a pixel on the screen can be done using Memory.peek and Memory.poke. The screen memory map of the Hack platform specifies that the pixel at column *col* and row *row* $(0 \le col \le 511, 0 \le row \le 255)$ is mapped to the *col* % 16 bit of memory location $16384 + row \cdot 32 + col/16$. Drawing a single pixel requires changing a single bit in the accessed word (and that bit only).

drawLine: The basic algorithm in figure 12.7 can potentially lead to overflow. However, the algorithm's improved version eliminates the problem.

Some aspects of the algorithm should be generalized for drawing lines that extend to four possible directions. Be reminded that the screen origin (coordinates (0,0)) is at the top-left corner. Therefore, some of the directions and plus/minus operations specified in the algorithm should be modified by your drawLine implementation.

The special yet frequent cases of drawing straight lines, that is, when $dx = 0$ or $dy = 0$, should not be handled by this algorithm. Rather, they should benefit from a separate and optimized implementation.

drawCircle: The algorithm shown in figure 12.8 can potentially lead to overflow. Limiting circle radii to be at most 181 is a reasonable solution.

Output

The Output class is a library of functions for displaying characters. The class assumes a character-oriented screen consisting of 23 rows (indexed 0 … 22, top to bottom) of 64 characters each (indexed 0 … 63, left to right). The top-left character location on the screen is indexed (0,0). A visible cursor, implemented as a small filled square, indicates where the next character will be displayed. Each character is displayed by rendering on the screen a rectangular image, 11 pixels high and 8 pixels wide (which includes margins for character spacing and line spacing). The collection of all the character images is called a *font*.

Font implementation: The design and implementation of a font for the Hack character set (appendix 5) is a drudgery, combining artistic judgment and rote implementation work. The resulting font is a collection of ninety-five rectangular bitmap images, each representing a printable character.

Fonts are normally stored in external files that are loaded and used by the character-drawing package, as needed. In Nand to Tetris, the font is embedded in the OS Output class. For each printable character, we define an array that holds the character's bitmap. The array consists of 11 elements, each corresponding to a row of 8 pixels. Specifically, we set the value of each array entry *j* to an integer value whose binary representation (bits) codes the 8 pixels appearing in the *j*-th row of the character's bitmap. We also define a static array of size 127, whose index values 32 … 126 correspond to the codes of the printable characters in the Hack character set (entries 0 … 31 are not used). We then set each array entry *i* of that array to the 11-entry array that represents the bitmap image of the character whose character code is *i* (did we mention drudgery?).

The project 12 materials include a skeletal Output class containing Jack code that carries out all the implementation work described above. The given code implements the ninety-five-character font, except for one character, whose design and implementation is left as an exercise. This code can be activated by the Output.init function, which can also initialize the cursor.

printChar: Displays the character at the cursor location and advances the cursor one column forward. To display a character at location (*row*,*col*), where $0 \le row \le 22$ and $0 \le col \le 63$, we write the character's bitmap onto the box of pixels ranging from $11 \cdot row$ to $11 \cdot row + 10$ and from $8 \cdot col$ to $8 \cdot col + 7$.

printString: Can be implemented using a sequence of printChar calls.

printInt: Can be implemented by converting the integer to a string and then printing the string.

Keyboard

The Hack computer memory organization (see section 5.2.6) specifies that the *keyboard memory map* is a single 16-bit memory register located at address 24576.

keyPressed: Can be implemented easily using Memory.peek ().

readChar, readString: Can be implemented by following the algorithms in figure 12.10.

readInt: Can be implemented by reading a string and converting it into an int value using a String method.

Sys

wait: This function is supposed to wait a given number of milliseconds and return. It can be implemented by writing a loop that runs approximately duration milliseconds before terminating. You will have to time your specific computer to obtain a one millisecond wait, as this constant varies from one CPU to another. As a result, your Sys.wait() function will not be portable. The function can be made portable by running yet another configuration function that sets various constants reflecting the hardware specifications of the host platform, but for Nand to Tetris this is not needed.

halt: Can be implemented by entering an infinite loop.

init: According to the Jack language specification (see section 9.2.2), a Jack program is a set of one or more classes. One class must be named Main, and this class must include a function named main. To start running a program, the Main.main function should be called.

The operating system is also a collection of compiled Jack classes. When the computer boots up, we want to start running the operating system and have *it* start running the main program. This chain of command is implemented as follows. According to the Standard VM Mapping on the Hack Platform (section 8.5.2), the VM translator writes bootstrap code (in machine language) that calls the OS function Sys.init. This bootstrap code is stored in the ROM, starting at address 0. When we reset the computer, the program counter is set to 0, the bootstrap code starts running, and the Sys.init function is called.

With that in mind, Sys.init should do two things: call all the init functions of the other OS

classes, and then call Main.main.

From this point onward the user is at the mercy of the application program, and the Nand to Tetris journey has come to an end. We hope that you enjoyed the ride!

12.4 Project

Objective: Implement the operating system described in the chapter.

Contract: Implement the operating system in Jack, and test it using the programs and testing scenarios described below. Each test program uses a subset of OS services. Each of the OS classes can be implemented and unit-tested in isolation, in any order.

Resources: The main required tool is Jack—the language in which you will develop the OS. You will also need the supplied Jack compiler for compiling your OS implementation, as well as the supplied test programs, also written in Jack. Finally, you'll need the supplied VM emulator, which is the platform on which the tests will be executed.

Your projects/12 folder includes eight skeletal OS class files named Math.jack, String.jack, Array.jack, Memory.jack, Screen.jack, Output.jack, Keyboard.jack, and Sys.jack. Each file contains the signatures of all the class subroutines. Your task is completing the missing implementations.

The VM emulator: Operating system developers often face the following chicken-and-egg dilemma: How can we possibly test an OS class in isolation, if the class uses the services of other OS classes not yet developed? As it turns out, the VM emulator is perfectly positioned to support unit-testing the OS, one class at a time.

Specifically, the VM emulator features an executable version of the OS, written in Java. When a VM command call foo is found in the loaded VM code, the emulator proceeds as follows. If a VM function named foo exists in the loaded code base, the emulator executes its VM code. Otherwise, the emulator checks whether foo is one of the built-on OS functions. If so, it executes foo's built-in implementation. This convention is ideally suited for supporting the testing strategy that we now turn to describe.

Testing Plan

Your projects/12 folder includes eight test folders, named MathTest, MemoryTest, …, for testing each one the eight OS classes Math, Memory, …. Each folder contains a Jack program, designed to test (by using) the services of the corresponding OS class. Some folders contain test scripts and compare files, and some contain only a .jack file or files. To test your implementation of the OS class *Xxx*.jack, you may proceed as follows:

- Inspect the supplied *Xxx*Test/*.jack code of the test program. Understand which OS services

are tested and how they are tested.

- Put the OS class *Xxx*.jack that you developed in the *Xxx*Test folder.
- Compile the folder using the supplied Jack compiler. This will result in translating both your OS class file and the .jack file or files of the test program into corresponding .vm files, stored in the same folder.
- If the folder includes a .tst test script, load the script into the VM emulator; otherwise, load the folder into the VM emulator.
- Follow the specific testing guidelines given below for each OS class.

Memory, Array, Math: The three folders that test these classes include test scripts and compare files. Each test script begins with the command load. This command loads all the .vm files in the current folder into the VM emulator. The next two commands in each test script create an output file and load the supplied compare file. Next, the test script proceeds to execute several tests, comparing the test results to those listed in the compare file. Your job is making sure that these comparisons end successfully.

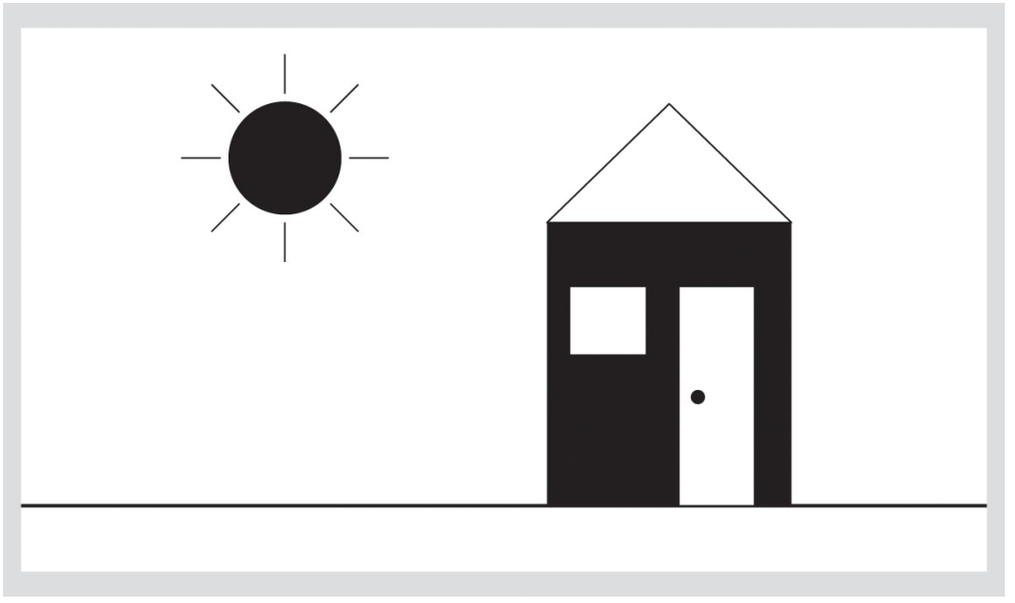
Note that the supplied test programs don't comprise a full test of Memory.alloc and Memory.deAlloc. A complete test of these memory management functions requires inspecting internal implementation details not visible in user-level testing. If you want to do so, you can test these functions by using step-by-step debugging and by inspecting the state of the host RAM.

String: Execution of the supplied test program should yield the following output:

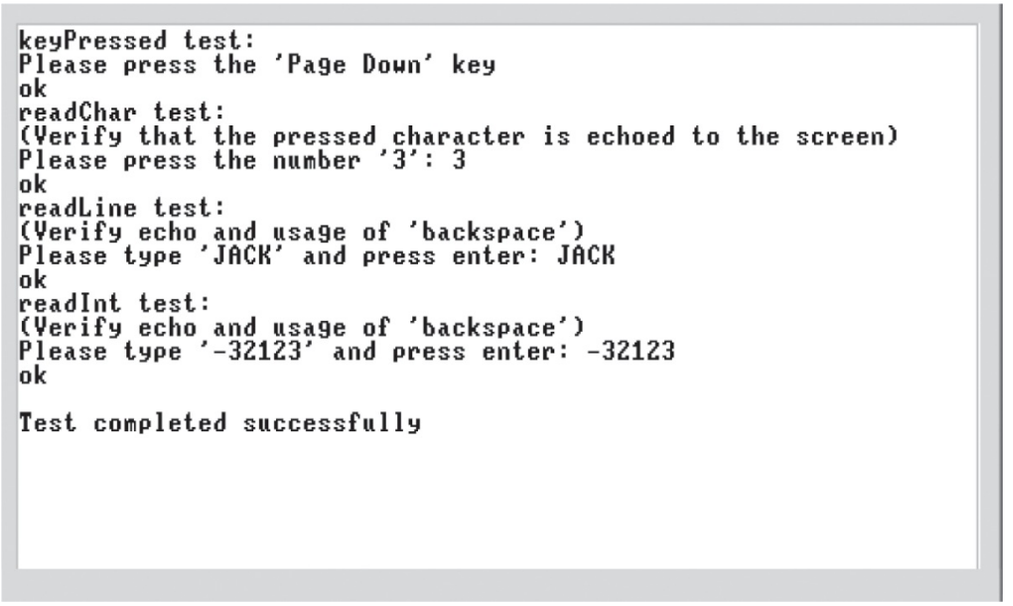
```
new,appendChar: abcde
setInt: 12345
setInt: -32767
length: 5
charAt[2]: 99
setCharAt(2,'-'): ab-de
eraseLastChar: ab-d
intValue: 456
intValue: -32123
backSpace: 129
doubleQuote: 34
newLine: 128
```

Output: Execution of the supplied test program should yield the following output:

Screen: Execution of the supplied test program should yield the following output:



Keyboard: This OS class is tested by a test program that effects user-program interaction. For each function in the Keyboard class (keyPressed, readChar, readLine, readInt) the program prompts the user to press some keys. If the OS function is implemented correctly and the requested keys are pressed, the program prints ok and proceeds to test the next OS function. Otherwise, the program repeats the request. If all requests end successfully, the program prints Test ended successfully. At this point the screen should show the following output:



Sys

The supplied .jack file tests the Sys.wait function. The program requests the user to press a key (any key) and waits two seconds, using a call to Sys.wait. It then prints a message on the screen. Make sure that the time that elapses between your release of the key and the appearance of the printed message is about two seconds.

The Sys.init function is not tested explicitly. However, recall that it performs all the necessary OS initializations and then calls the Main.main function of each test program. Therefore, we can assume that nothing will work properly unless Sys.init is implemented correctly.

Complete Test

After testing successfully each OS class in isolation, test your entire OS implementation using the Pong game introduced earlier in the book. The source code of Pong is available in projects/11/Pong. Put your eight OS .jack files in the Pong folder, and compile the folder using the supplied Jack compiler. Next, load the Pong folder into the VM emulator, execute the game, and ensure that it works as expected.

12.5 Perspective

This chapter presented a subset of basic services that can be found in most operating systems. For example, managing memory, driving I/O devices, supplying mathematical operations not implemented in hardware, and implementing abstract data types like the string abstraction. We have chosen to call this standard software library an *operating system* to reflect its two

main functions: encapsulating the gory hardware details, omissions, and idiosyncrasies in transparent software services, and enabling compilers and application programs to use these services via clean interfaces. However, the gap between what we have called an OS and industrial-strength operating systems remains wide.

For starters, our OS lacks some of the basic services most closely associated with operating systems. For example, our OS supports neither multi-threading nor multiprocessing; in contrast, the kernel of most operating systems is devoted to exactly that. Our OS supports no mass storage devices; in contrast, the main data store handled by operating systems is a file system abstraction. Our OS features neither a command-line interface (as in a Unix shell) nor a graphical interface consisting of windows and menus. In contrast, this is the operating system interface that users expect to see and interact with. Numerous other services commonly found in operating systems are not present in our OS, like security, communication, and more.

Another notable difference lies in the liberal way in which our OS operations are invoked. Some OS operations, for example, peek and poke, give the programmer complete access to the host computer resources. Clearly, inadvertent or malicious use of such functions can cause havoc. Therefore, many OS services are considered privileged, and accessing them requires a security mechanism that is more elaborate than a simple function call. In contrast, in the Hack platform, there is no difference between OS code and user code, and operating system services run in the same user mode as that of application programs.

In terms of efficiency, the algorithms that we presented for multiplication and division were standard. These algorithms, or variants thereof, are typically implemented in hardware rather than in software. The running time of these algorithms is $O(n)$ addition operations. Since adding two n-bit numbers requires $O(n)$ bit operations (gates in hardware), these algorithms end up requiring $O(n^2)$ bit operations. There exist multiplication and division algorithms whose running time is asymptotically significantly faster than $O(n^2)$, and for a large number of bits these algorithms are more efficient. In a similar fashion, optimized versions of the geometric operations that we presented, like line drawing and circle drawing, are often implemented in special graphics-acceleration hardware.

Like every hardware and software system developed in Nand to Tetris, our goal is not to provide a complete solution that addresses all wants and needs. Rather, we strive to build a working implementation and a solid understanding of the system's *foundation*, and then propose ways to extend it further. Some of these optional extension projects are mentioned in the next and final chapter in the book.