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 previous chapters of this book, we described and built the hardware architecture of a computer platform, called *Hack*, and the software hierarchy that makes it usable. In particular, we introduced an object-based language, called *Jack*, and described how to write a compiler for it. Other high-level programming languages can be specified on top of the Hack platform, each requiring its own compiler.

The last major interface missing in this puzzle is an *operating system* (OS). The OS is designed to close gaps between the computer's hardware and software systems, and to make the overall computer more accessible to programmers and users. For example, in order to render the text "`Hello World`" on our computer's screen, several hundred pixels must be drawn at specific screen locations. This can be done by consulting the hardware specification and writing code that puts the necessary bits in the RAM-resident screen memory map. Obviously, high-level programmers expect something better than that. They want to use a command like `printString("Hello World")` and let someone else worry about the details. And that's where the operating system enters the picture.

Throughout this chapter, the term *operating system* is used rather loosely. In fact, the OS services that we describe comprise an operating system in a very minimal fashion, aiming at (i) encapsulating various hardware-specific services in a software-friendly way, and (ii) extending high-level languages with various functions and abstract data types. The dividing line between an operating system in this sense and a standard language library is not very clear. Indeed, some modern languages, most notably Java, tend to pack many classic operating system services like GUI management, memory management, and multitasking in its standard software library, along with many language extensions.

Following this pattern, the collection of services that we specify and build in this chapter can be viewed as a combination of a simple OS and a standard library for the Jack language. This OS is packaged as a collection of Jack classes, each providing a set of related services via Jack subroutine calls. The resulting OS has many features resembling those of industrial strength operating systems, but it still lacks numerous OS features such as process handling, disk management, communications, and more.

Operating systems are usually written in a high-level language and compiled into binary form, just like any other program. Our OS is no exception—it can be written completely in Jack. Yet unlike other programs written in high-level languages, the operating system code must be aware of the hardware platform on which it runs. In other words, in order to hide the gory hardware details from the application programmer, the OS programmer must write code that manipulates these details directly (a task that requires access to the hardware documentation). Conveniently, this can be done using the Jack language. As we observe in this chapter, Jack was defined with sufficient "lowness" in it, permitting an intimate closeness to the hardware when needed.

The chapter starts with a relatively long Background section, describing key algorithms normally used to implement basic operating system services. These include mathematical functions, string operations, memory management, handling text and graphics output to the screen, and handling inputs from the keyboard. This algorithmic introduction is followed by a Specification section, providing the complete API of the Jack OS, and an Implementation section, describing how to build the OS using the classic algorithms presented earlier. As usual, the final Project section provides all the necessary project materials for gradual construction and unit-testing the entire OS presented in the chapter.

The chapter provides two key lessons, one in software engineering and one in computer science. First, we complete the construction of the high-level language, compiler, and operating system trio. Second, since operating system services must execute efficiently, we pay attention to running time considerations. The result is an elegant series of algorithms, each being a computer science gem.

12.1 Background

12.1.1 Mathematical Operations

Computer systems must support mathematical operations like addition, multiplication, and division. Normally, addition is implemented in hardware, at the ALU

level, as we have done in chapter 3. Other operations like multiplication and division can be handled by either hardware or software, depending on the computer's cost/performance requirements. This section shows how multiplication, division, and square root operations can be implemented efficiently in software, at the OS level. We note in passing that hardware implementations of these mathematical operations can be based on the same algorithms presented here.

Efficiency First Mathematical algorithms operate on n-bit binary numbers, with typical computer architectures having $n = 16, 32,$ or 64. As a rule, we seek algorithms whose running time is proportional (or at least polynomial) in this parameter n. Algorithms whose running time is proportional to the *value* of n-bit numbers are unacceptable, since these values are exponential in n. For example, suppose we implement the multiplication operation $x \cdot y$ using the repeated addition algorithm *for* $i = 1 \ldots y$ $\{result = result + x\}$. Well, the problem is that in a 64-bit computer, y can be greater than 18,000,000,000,000,000,000, implying that this naïve algorithm may run for years even on the fastest computers. In sharp contrast, the running time of the multiplication algorithm that we present below is proportional not to the multiplicands' value, which may be as large as 2^n, but rather to n. Therefore, it will require only $c \cdot n$ elementary operations for *any pair of multiplicands*, where c is a small constant representing the number of elementary operations performed in each loop iteration.

We use the standard "Big-Oh" notation, $O(n)$, to describe the running time of algorithms. Readers who are not familiar with this notation can simply read $O(n)$ as "in the order of magnitude of n." With that in mind, we now turn to present an efficient multiplication $x \cdot y$ algorithm for n-bit numbers whose running time is $O(n)$ rather than $O(x)$ or $O(y)$, which are exponentially larger.

Multiplication Consider the standard multiplication method taught in elementary school. To compute 356 times 27, we line up the two numbers one on top of the other. Next, we multiply each digit of 356 by 7. Next, we "shift to the left" one position, and multiply each digit of 356 by 2. Finally, we sum up the columns and obtain the result. The binary version of this technique—figure 12.1—follows exactly the same logic.

The algorithm in figure 12.1 performs $O(n)$ addition operations on n-bit numbers, where n is the number of bits in x and y. Note that $shiftedX * 2$ can be efficiently obtained by either left-shifting its bit representation or by adding *shiftedX* to itself. Both operations can be easily performed using primitive ALU operations. Thus this algorithm lends itself naturally to both software and hardware implementations.

Long multiplication

x			1	0	1	1	=	1	1	
y		$*$		1	0	1	=		5	j-th bit of y
			1	0	1	1				1
		0	0	0	0					0
	1	0	1	1						1
$x \cdot y$	1	1	0	1	1	1	=	5	5	

```
multiply(x, y):
  // Where x, y ≥ 0
  sum = 0
  shiftedX = x
  for j = 0 . . . (n − 1) do
      if (j-th bit of y) = 1 then
          sum = sum + shiftedX
      shiftedX = shiftedX ∗ 2
```

Figure 12.1 Multiplication of two n-bit numbers.

A Comment about Notation The algorithms in this chapter are written using a self-explanatory pseudocode syntax. The only non-obvious convention is that we use indentation to represent blocks of code (avoiding curly brackets or begin/end keywords). For example, in figure 12.1, $sum = sum + shiftedX$ belongs to the single-statement body of the *if* statement whereas $shiftedX = shiftedX * 2$ ends the two-statement body of the *for* statement.

Division The naïve way to compute the division of two n-bit numbers x/y is to repeatedly subtract y from x until it is impossible to continue (i.e., until $x < y$). The running time of this algorithm is clearly proportional to the quotient, and may be as large as $O(x)$, that is, exponential in the number of bits n. To speed up this algorithm, we can try to subtract large chunks of y's from x in each iteration. For example, if $x = 891$ and $y = 5$, we can tell right away that we can deduct a hundred 5's from x and the remainder will still be greater than 5, thus shaving 100 iterations from the naïve approach. Indeed, this is the rationale behind the school method for long division x/y. Formally, in each iteration we try to subtract from x the largest possible shift of y, namely, $y \cdot T$ where T is the largest power of 10 such that $y \cdot T \leq x$.

```
divide (x, y):
  // Integer part of x/y, where x ≥ 0 and y > 0
  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.

The binary version of this opportunistic algorithm is identical, except that T is a power of 2 instead of 10.

Writing down this long division algorithm as we have done for multiplication is an easy exercise. We find it more illuminating to formulate the same logic as a recursive program that is probably easier to implement, shown in figure 12.2.

The running time of this algorithm is determined by the depth of the recursion. Since in each level of recursion the value of y is multiplied by 2, and since we terminate once $y > x$, it follows that the recursion depth is bounded by n, the number of bits in x. Each recursion level involves a constant number of addition, subtraction, and multiplication operations, implying a total running time of $O(n)$ such operations.

This algorithm may be considered suboptimal since each multiplication operation also requires $O(n)$ addition and subtraction operations. However, careful inspection reveals that the product $2 \cdot q \cdot y$ can be computed without any multiplication. Instead, we can rely on the value of this product in the previous recursion level, and use addition to establish its current value.

Square Root Square roots can be computed efficiently in a number of different ways, for example, by 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 convenient properties. First, it is monotonically increasing. Second, its inverse function, $x = y^2$, is something that we already know how to compute (multiplication). Taken together, these properties imply that we have all we need to compute square roots using *binary search*. Figure 12.3 gives the details.

Note that each loop iteration takes a constant number of arithmetic operations. Since the number of iterations is bound by $n/2$, the algorithm's running time is $O(n)$ arithmetic operations.

sqrt(x):
// Compute the integer part of $y = \sqrt{x}$. Strategy:
// Find an integer y such that $y^2 \le x < (y+1)^2$ (for $0 \le x < 2^n$)
// By performing a binary search in the range $0 \ldots 2^{n/2} - 1$.
$y = 0$
for $j = n/2 - 1 \ldots 0$ do
  if $(y + 2^j)^2 \le x$ then $y = y + 2^j$
return y

Figure 12.3 Square root computation using binary search.

12.1.2 String Representation of Numbers

Computers represent numbers internally using binary codes. Yet humans are used to dealing with numbers in a decimal notation. Thus, when humans have to read or input numbers, and only then, a conversion to or from decimal notation must be performed. Typically, this service is implicit in the character handling routines supplied by the operating system. We now turn to describe how these OS services are actually implemented.

Of course the only subset of characters which is of interest here are the ten digit symbols that represent actual numbers. The ASCII codes of these characters are as follows:

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

As gleaned from the ASCII code, single digit characters can be easily converted into their numeric representation, and vice versa, as follows. To compute the ASCII code of a given digit $0 \le x \le 9$, we can simply add x to 48 – the code of `'0'`. Conversely, the numeric value represented by an ASCII code $48 \le c \le 57$ is obtained by $c - 48$. And once we know how to convert single digits, we can proceed to convert any given integer. These conversion algorithms can be based on either iterative or recursive logic, so we present one of each in figures 12.4 and 12.5, respectively.

12.1.3 Memory Management

Dynamic Memory Allocation Computer programs declare and use all sorts of variables, including simple data items like integers and booleans and complex ones like arrays and objects. One of the greatest virtues of high-level languages is that pro-

```
// Convert a non-negative number to a string
int2String(n):
  lastDigit = n % 10
  c = character representing lastDigit
  if n < 10
    return c (as a string)
  else
    return int2String(n/10).append(c)
```

```
// Convert a string to a non-negative number
string2Int(s):
  v = 0
  for i = 1 ... length of s do
    d = integer value of the digit s[i]
    v = v * 10 + d
  return v
  // (Assuming that s[1] is the most
  //  significant digit character of s.)
```

Figures 12.4 and 12.5 String-numeric conversions.

grammers don't have to worry about the details of allocating RAM space to these variables and recycling the space when it is no longer needed. Instead, all these memory management chores are done behind the scene by the compiler, the operating system, and the virtual machine implementation. This section describes the role of the operating system in this joint effort.

Different variables are allocated memory at different points of time during the program's life cycle. For example, *static variables* may be allocated by the compiler at compile time, while *local variables* are allocated on the stack each time a subroutine starts running. Other memory is dynamically allocated during the program's execution, and that's where the OS enters the picture. For example, each time a Java program creates a new array or a new object, a memory block whose size can be determined only during run-time should be allocated. And when the array or the object is no longer needed, its RAM space may be recycled. In some languages like C++ and Jack, de-allocation of un-needed space is the responsibility of the programmer, while in others, like Java, "garbage collection" occurs automatically. The RAM segment from which memory is dynamically allocated is called *heap*, and the agent responsible for managing this resource is the operating system.

Operating systems use various techniques for handling dynamic memory allocation and de-allocation. These techniques are implemented in two functions traditionally called `alloc()` and `deAlloc()`. We present two versions of these algorithms: a basic one and an improved one.

Basic Memory Allocation Algorithm The data structure that this algorithm manages is a single pointer, called *free*, which points to the beginning of the heap segment that was not yet allocated. Figure 12.6a gives the details.

```
Initialization: free = heapBase

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

// De-allocate the memory space of a given object.
deAlloc(object):
   do nothing
```

Figure 12.6a Basic memory allocation scheme (wasteful).

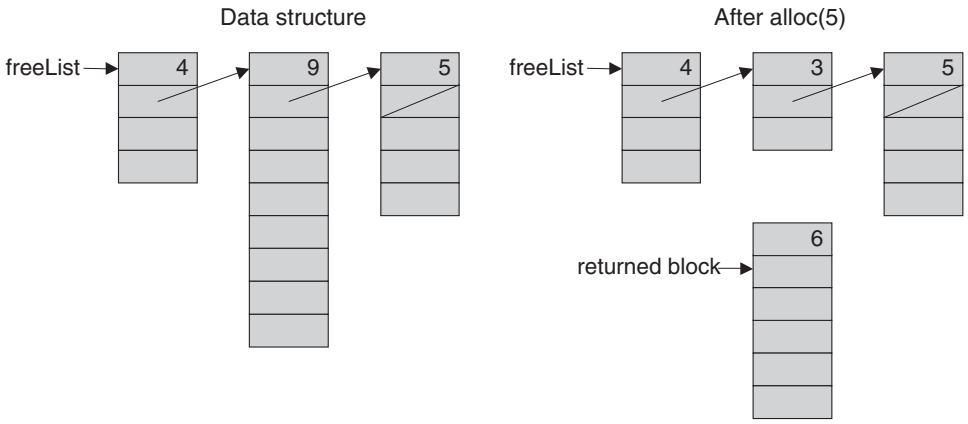
This algorithm is clearly wasteful, as it does not reclaim the space of decommissioned objects.

Improved Memory Allocation Algorithm This algorithm manages a linked list of available memory segments, called *freeList*. Each segment contains two housekeeping fields: the segment's length and a pointer to the next segment in the list. These fields can be physically kept in the segment's first two memory locations. For example, the implementation can use the convention `segment.length==segment[0]` and `segment.next==segment[1]`. Figure 12.6b (top left) illustrates a typical *freeList* state.

When asked to allocate a memory block of some given size, the algorithm has to search the *freeList* for a suitable segment. There are two well-known heuristics for doing this search. *Best-fit* finds the segment whose length is the closest (from above) to 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 taken from it (the location just before the beginning of the returned block, `block[-1]`, is reserved to hold its length, to be used during de-allocation). Next, this segment is updated in the *freeList*, becoming the part that remained after the allocation. If no memory was left in the block, 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 de-allocated block to the *freeList*. The details are given in figure 12.6b.

After a while, dynamic memory allocation schemes like the algorithm in figure 12.6b may create a block fragmentation problem. Hence, some kind of "defrag" op-



Initialization:
freeList = *heapBase*
freeList.length = *heapLength*
freeList.next = null

// Allocate a memory space of *size* words.
alloc(*size*):
Search *freeList* using best-fit or first-fit heuristics
to obtain a segment with *segment.length* > *size*
If no such segment is found, return failure
(or attempt defragmentation)
block = needed part of the found segment
(or all of it, if the segment remainder is too small)
Update *freeList* to reflect the allocation
block[−1] = *size* + 1 // Remember block size, for de-allocation
Return *block*

// Deallocate a decommissioned *object*.
deAlloc(*object*):
segment = *object* − 1
segment.length = *object*[−1]
Insert *segment* into the *freeList*

Figure 12.6b Improved memory allocation scheme (with recycling).

eration should be considered, namely, merging memory areas that are physically consecutive in memory but logically split into different segments in the *freeList*. The defragmentation operation can be done each time an object is de-allocated, or when `alloc()` fails to find an appropriate block, or according to some other intermediate or ad-hoc condition.

12.1.4 Variable-length Arrays and Strings

Suppose we want to use high-level operations like `s1="New York"` or `s2=readLine("enter a city")`. How can we implement these variable-length abstractions? The common approach in modern languages is to use a `String` class that supplies services for creating and manipulating string objects. The string object can be physically realized using an array. Normally, when the string is created, this array is allocated to hold some maximum possible length. The actual length of the string at each point of time may be shorter than this maximum, and must be maintained throughout the string object's life cycle. For example, if we issue a command like `s1.eraseLastChar()`, the actual length of `s1` should decrease from 8 to 7 (although the length of the initially created array does not change). In general then, array locations beyond the current length are not considered part of the string contents.

Most programming languages feature string types, as well as other data types of variable lengths. The string objects are usually provided by the language's standard library, for example, the `String` and `StringBuffer` classes in Java or the `strXXX` functions in C.

12.1.5 Input/Output Management

Computers are typically connected to a variety of input/output devices such as keyboard, screen, mouse, disk, network card, etc. Each of these I/O devices has its own electromechanical and physical idiosyncrasies, and thus reading and writing data on them involves many technical details. High-level languages abstract these details away from the programmer using high-level operations like `c=readChar()` and `printChar(c)`. These operations are implemented by OS routines that carry out the actual I/O.

Hence, an important function of the operating system is handling the various I/O devices connected to the computer. This is done by encapsulating the details of interfacing the device and by providing convenient access to its basic functionality, using a set of O/S routines collectively known as the *device driver*. In this book we describe the basic elements of handling the two most prevalent I/O devices: a screen

and a keyboard. We divide the handling of the screen into two logically separate modules: handling *graphics output* and handling *character output*.

Graphics Output

Pixel Drawing Most computers today use *raster*, also called *bitmap*, display technologies. The only primitive operation that can be physically performed in a bitmap screen is drawing an individual *pixel*—a single "dot" on the screen specified by (*column*, *row*) coordinates. The usual convention is that columns are numbered from left to right (like the conventional x-axis) while rows are numbered from the top down (opposite of the conventional y-axis). Thus the screen coordinates of the top left pixel are (0,0).

The low-level drawing of a single pixel is a hardware-specific operation that depends on the particular interface of the screen and the underlying graphics card. If the screen interface is based on a RAM-resident *memory map*, as in Hack, then drawing a pixel is achieved by writing the proper binary value into the RAM location that represents the required pixel in memory (see figure 12.7).

The memory map interface of the Hack screen was described in section 5.2.4. Formulating a `drawPixel` algorithm that follows this contract is a simple task left to the reader as an exercise. So, now that we know how to draw a single pixel, let us turn to describing how to draw lines and circles.

Line Drawing When asked to draw a line between two locations on a bitmap screen, 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. Note that the "pen" that we use 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. Since the receptor cells in the human eye's retina also form a grid of "input pixels," there is a limit to the image granularity that the human eye can resolve anyway. Thus, high-resolution screens and printers can fool the human eye to

drawPixel (x, y):
// Hardware-specific.
// Assuming a memory mapped screen:
Write a predetermined value in the RAM location corresponding to screen location (x, y).

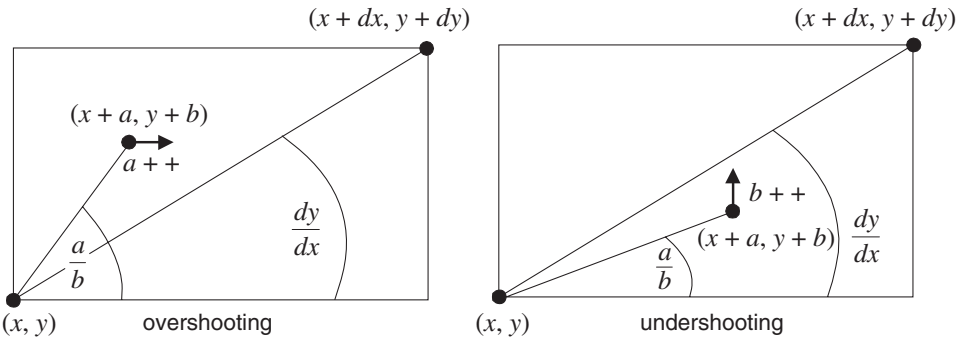
Figure 12.7 Drawing a pixel.

believe that the lines drawn by pixels or printed dots are visibly smooth. In fact they are always jagged.

The procedure for drawing a line from location $(x1, y1)$ to location $(x2, y2)$ starts by drawing the $(x1, y1)$ pixel and then zigzagging in the direction of $(x2, y2)$, until this pixel is reached. See figure 12.8a for the details.

To extend this algorithm to a general-purpose line drawing routine, one also has to take care of the possibilities $dx, dy < 0$, $dx > 0$, $dy < 0$, and $dx < 0$, $dy > 0$. To complete the picture, note that the special cases $dx = 0$ or $dy = 0$, required for drawing vertical and horizontal lines, are not handled by this algorithm. These widely used cases should probably benefit from a separate and optimized treatment anyway.

An annoying feature of the algorithm in figure 12.8a is the use of division operations in each loop iteration. Not only are these division operations time-consuming, but they also require floating point operations rather than simple integer arithmetic. The first obvious solution is to replace the $a/dx < b/dy$ condition with the equivalent $a \cdot dy < b \cdot dx$, which requires only integer multiplication. Further, careful inspection of the algebraic structure of the latter condition reveals that it may be checked




```
drawLine(x, y, x + dx, y + dy):
  // Assuming dx, dy > 0
  initialize (a, b) = (0, 0)
  while a ≤ dx and b ≤ dy do
    drawPixel(x + a, y + b)
    if a/dx < b/dy then a++ else b++
```

Figure 12.8a Line drawing.

// To test whether $a/dx < b/dy$, maintain a variable *adyMinusbdx*,
// and test if it becomes negative.
Initialization: set $adyMinusbdx = 0$
When $a{+}{+}$ is performed: set $adyMinusbdx = adyMinusbdx + dy$
When $b{+}{+}$ is performed: set $adyMinusbdx = adyMinusbdx - dx$

Figure 12.8b Efficient testing using addition operations only.

without using any multiplication at all. As shown in figure 12.8b, this may be done efficiently by maintaining a variable that updates the value of $a \cdot dy - b \cdot dx$ each time either a or b are modified.

Circle Drawing There are several ways to draw a circle on a bitmap screen. We present an algorithm (figure 12.9) that uses three routines already implemented in this chapter: *multiplication*, *square root*, and *line drawing*.

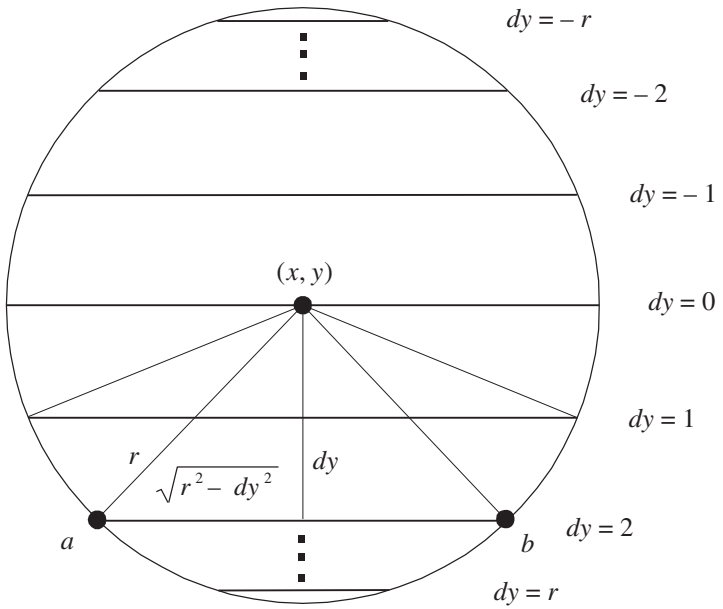
The algorithm is based on drawing a series of horizontal lines (like the typical line *ab* in figure 12.9), one for each row in the range $y - r$ to $y + r$. Since r is specified in pixels, the algorithm ends up drawing a line in every screen row along the circle's north-south axis, resulting in a completely filled circle. A trivial tweaking of this algorithm can yield an empty circle as well.

Note that the algorithm is somewhat inefficient, since the square root computation in each iteration is an expensive operation. There exist many more efficient circle-drawing algorithms, including ones that involve addition operations only, in the same spirit of our line-drawing algorithm.

Character Output All the output that we have described so far is graphical: pixels, lines, and circles. We now describe how *characters* are printed on the screen, pixel by pixel, using the good services of the operating system. Here are the details.

To develop a capability to write text on a bitmap screen, we first have to divide the physical pixel-oriented screen into a logical, character-oriented screen suitable for writing complete characters. For example, consider a screen that is 256 rows by 512 columns. If we allocate a grid of $11 * 8$ pixels for drawing a single character (11 rows, 8 columns), then our screen can show 23 lines of 64 characters each (with 3 extra rows of pixels left unused).

Next, for each character that we want to display on the screen, we can design a good-looking *font*, and then implement the font using a series of character bitmaps. For example, figure 12.10 gives a possible bitmap for the letter `'A'`.



drawCircle(x, y, r):
  for each $dy \in -r \ldots r$ do
    drawLine from $(x - \sqrt{r^2 - dy^2}, y + dy)$ to $(x + \sqrt{r^2 - dy^2}, y + dy)$

Figure 12.9 Circle drawing.

Note that in order for our display scheme to account for the requisite inter-character spacing, we must make sure that the $11 * 8$ bitmap of each character includes at least a 1-pixel space before the next character and at least a 1-pixel space between adjacent lines (the exact spacing may vary with the size of the individual characters).

Characters are usually drawn on the screen one after the other, from left to right. For example, the two commands `print("a")` and `print("b")` probably mean that the programmer wants to see the image `"ab"` drawn on the screen. Thus the character-writing package must maintain a ''cursor'' object that keeps track of the screen location where the next character should be drawn. The cursor information

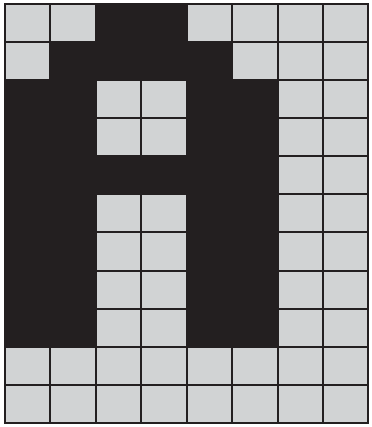


Figure 12.10 Character bitmap of the letter "A".

consists of *line* and *column* counts. For example, the character screen described at the section's beginning is characterized (excuse the pun) by $0 \leq line \leq 22$ and $0 \leq column \leq 63$. Drawing a single character at location (*line*, *column*) is achieved by writing the character's bitmap onto the box of pixels at rows $line \cdot 11 \ldots line \cdot 11 + 10$ and columns $column \cdot 8 \ldots column \cdot 8 + 7$. After the character has been drawn, the cursor should be moved one step to the right (i.e., $column = column + 1$), and, when a new line is requested, *row* should be increased by 1 and *column* reset to 0. When the bottom of the screen is reached, there is a question of what to do next, the common solution being to effect a "scrolling" operation. Another possibility is starting over at the top left corner, namely, setting the cursor to (0,0).

To conclude, we know how to write characters on the screen. Writing other types of data follows naturally from this basic capability: *strings* are written character by character, *numbers* are first converted to strings and then written as strings, and so on.

Keyboard Handling Handling user-supplied text input is more involved than meets the eye. For example, consider the command `name=readLine("enter your name:")`. The low-level implementation of this command is not trivial, since it involves an unpredictable event: A human user is supposed to press some keys on the keyboard before this code can terminate properly. And the problem, of course, is that human users press keyboard keys for variable durations of time. Hence, the trick is to encapsulate the handling of all these messy low-level details in OS routines like `readLine`, freeing high-level programs from this tedium.

```
keyPressed( ):
  // Depends on the specifics of the keyboard interface
  if a key is presently pressed on the keyboard
    return the ASCII value of the key
  else
    return 0
```

Figure 12.11 Capturing "raw" keyboard input.

This section describes how the operating system manages text-oriented input in three increasing levels of abstraction: (i) detecting which key is currently pressed on the keyboard, (ii) capturing single-character inputs, and (iii) capturing multi-character inputs, that is, strings.

Detecting Keyboard Input In the lowest-level form of capturing keyboard input, the program gets data directly from the hardware, indicating which key is currently pressed by the user. The access to this raw data depends on the specifics of the keyboard interface. For example, if the interface is a *memory map* that is continuously refreshed from the keyboard, as in Hack, we can simply inspect the contents of the relevant RAM area to determine which key is presently pressed. The details of this inspection can then be incorporated into the implementation of the algorithm in figure 12.11.

For example, if you know the RAM address of the keyboard memory map in the host computer, the implementation of this algorithm entails nothing more than a memory lookup.

Reading a Single Character The elapsed time between "key pressed" and "key released" events is unpredictable. Hence, we have to write code that neutralizes this variation. Also, when users press keys on the keyboard, we usually want to give a visual 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 typically want to echo the inputted character by displaying its bitmap on the screen at the cursor location. This logic is implemented in figure 12.12.

Reading a String Usually, a multi-key input typed by the user is considered final only after the `enter` key has been pressed, yielding the *newline* character. And, until

```
readChar( ):
  // Read and echo a single character
  display the cursor
  while no key is pressed on the keyboard
    do nothing // wait till a key is pressed
  c = code of currently pressed key
  while a key is pressed
    do nothing // wait for the user to let go
  print c at the current cursor location
  move the cursor one position to the right
  return c
```

```
readLine( ):
  // Read and echo a "line" (until newline)
  s = empty string
  repeat
    c = readChar( )
    if c = newline character
      print newline
      return s
    else if c = backspace character
        remove last character from s
        move the cursor 1 position back
      else
        s = s.append(c)
```

Figures 12.12 and 12.13 Capturing "cooked" keyboard input.

the `enter` key is pressed, the user should be allowed to backspace and erase previously typed characters. The code that implements this logic and renders its visual effect is given in figure 12.13.

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 hardware.

12.2 The Jack OS Specification

The previous section presented a series of algorithms that address some classic operating system tasks. In this section we turn to formally specify one particular operating system—the Jack OS—in API form. Since the Jack OS can also be viewed as an extension of the Jack programming language, this documentation duplicates exactly "The Jack Standard Library" from section 9.2.7. In chapter 9, the OS specification was intended for programmers who want to use its abstract services; in this chapter, the OS specification is intended for programmers who have to implement these services. Technical information and implementation tips follow in section 12.3.

The operating system is divided into eight classes:

- *Math:* provides basic mathematical operations;
- *String:* implements the `String` type and string-related operations;
- *Array:* implements the `Array` type and array-related operations;
- *Output:* handles text output to the screen;
- *Screen:* handles graphic output to the screen;
- *Keyboard:* handles user input from the keyboard;
- *Memory:* handles memory operations;
- *Sys:* provides some execution-related services.

12.2.1 Math

This class enables various mathematical operations.

- function void **`init`**( ): for internal use only;
- function int **`abs`**(int x): returns the absolute value of x;
- function int **`multiply`**(int x, int y): returns the product of x and y;
- function int **`divide`**(int x, int y): returns the integer part of x/y;
- function int **`min`**(int x, int y): returns the minimum of x and y;
- function int **`max`**(int x, int y): returns the maximum of x and y;
- function int **`sqrt`**(int x): returns the integer part of the square root of x.

12.2.2 String

This class implements the `String` data type and various string-related operations.

- constructor String **`new`**(int maxLength): constructs a new empty string (of length zero) that can contain at most maxLength characters;
- method void **`dispose`**( ): disposes this string;
- method int **`length`**( ): returns the length of this string;
- method char **`charAt`**(int j): returns the character at location j of this string;
- method void **`setCharAt`**(int j, char c): sets the j-th element of this string to c;
- method String **`appendChar`**(char c): appends c to this string and returns this string;

- method void **eraseLastChar**( ): erases the last character from this string;
- method int **intValue**( ): returns the integer value of this string (or the string prefix until a non-digit character is detected);
- method void **setInt**(int j): sets this string to hold a representation of j;
- function char **backSpace**( ): returns the backspace character;
- function char **doubleQuote**( ): returns the double quote (“) character;
- function char **newLine**( ): returns the newline character.

12.2.3 Array

This class enables the construction and disposal of arrays.

- function Array **new**(int size): constructs a new array of the given size;
- method void **dispose**( ): disposes this array.

12.2.4 Output

This class allows writing text on the screen.

- function void **init**( ): for internal use only;
- function void **moveCursor**(int i, int j): moves the cursor to the j-th column of the i-th row, and erases the character displayed there;
- function void **printChar**(char c): prints c at the cursor location and advances the cursor one column forward;
- function void **printString**(String s): prints s starting at the cursor location and advances the cursor appropriately;
- function void **printInt**(int i): prints i starting at the cursor location and advances the cursor appropriately;
- function void **println**( ): advances the cursor to the beginning of the next line;
- function void **backSpace**( ): moves the cursor one column back.

12.2.5 Screen

This class allows drawing graphics on the screen. Column indices start at 0 and are left to right. Row indices start at 0 and are top to bottom. The screen size is hardware-dependant (in the Hack platform: 256 rows by 512 columns).

- function void **`init`**(): for internal use only;
- function void **`clearScreen`**(): erases the entire screen;
- function void **`setColor`**(boolean b): sets a color (white = false, black = true) to be used for all further `drawXXX` commands;
- function void **`drawPixel`**(int x, int y): draws the (x,y) pixel;
- function void **`drawLine`**(int x1, int y1, int x2, int y2): draws a line from pixel (x1,y1) to pixel (x2,y2);
- function void **`drawRectangle`**(int x1, int y1, int x2, int y2): draws a filled rectangle whose top left corner is (x1,y1) and whose bottom right corner is (x2,y2);
- function void **`drawCircle`**(int x, int y, int r): draws a filled circle of radius r <= 181 around (x,y).

12.2.6 Keyboard

This class allows reading inputs from a standard keyboard.

- function void **`init`**(): for internal use only;
- function char **`keyPressed`**(): returns the character of the currently pressed key on the keyboard; if no key is currently pressed, returns 0;
- function char **`readChar`**(): waits until a key is pressed on the keyboard and released, then echoes the key to the screen and returns the character of the pressed key;
- function String **`readLine`**(String message): prints the message on the screen, reads the line (text until a newline character is detected) from the keyboard, echoes the line to the screen, and returns its value. This function also handles user backspaces;
- function int **`readInt`**(String message): prints the message on the screen, reads the line (text until a newline character is detected) from the keyboard, echoes the line to the screen, and returns its integer value (until the first non-digit character in the line is detected). This function also handles user backspaces.

12.2.7 Memory

This class allows direct access to the main memory of the host platform.

- function void **`init`**(): for internal use only;
- function int **`peek`**(int address): returns the value of the main memory at this address;

- function void **poke**(int address, int value): sets the contents of the main memory at this address to value;
- function Array **alloc**(int size): finds and allocates from the heap a memory block of the specified size and returns a reference to its base address;
- function void **deAlloc**(Array o): De-allocates the given object and frees its memory space.

12.2.8 Sys

This class supports some execution-related services.

- function void **init**(): calls the `init` functions of the other OS classes and then calls the `Main.main()` function. For internal use only;
- function void **halt**(): halts the program execution;
- function void **error**(int errorCode): prints the error code on the screen and halts;
- function void **wait**(int duration): waits approximately *duration* milliseconds and returns.

12.3 Implementation

The operating system described in the previous section can be implemented as a collection of Jack classes. Each OS subroutine can be implemented as a Jack constructor, function, or method. The API of all these subroutines was given in section 12.2, and key algorithms were presented in section 12.1. This section provides some additional hints and suggestions for completing this implementation. Final technical details and test programs for unit-testing all the OS services are given in section 12.5. Note that most of the subroutines specified in the OS API are rather simple, requiring straightforward Jack programming. Thus we focus here only on the implementation of selected OS subroutines.

Some OS classes require class-level initialization. For example, some mathematical functions can run more quickly if they can use previously calculated values, kept in some static array, constructed once and for all in the `Math` class. As a rule, when an OS class `Xxx` needs some initialization code, this code should be embedded in a single function called `Xxx.init()`. Later in this section we explain how these `init()` functions are activated when the computer boots up and the OS starts running.

12.3.1 Math

`Math.multiply()`, `Math.divide()`: The algorithms in figures 12.1 and 12.2 are designed to operate on non-negative integers only. A simple way of handling negative numbers is applying the algorithms on absolute values and then setting the sign appropriately. For the multiplication algorithm, this is not really needed: it turns out that if the multiplicands are given in 2's complement, their product will be correct with no further ado.

Note that in each iteration j of the algorithm in figure 12.1, the j-th bit of the second number is extracted. We suggest encapsulating this operation in the following function:

`bit(x,j)`: Returns true if the j-th bit of the integer x is 1 and false otherwise.

The `bit(x,j)` function can be easily implemented using shifting operations. Alas, Jack does not support shifting. Instead, to speed up this function implementation in Jack, it may be convenient to define a fixed static array of length 16, say `twoToThe[j]`, whose j-th location holds the value 2 to the power of j. This array may be initialized once (in `Math.init`), and then used, via bitwise Boolean operations, in the implementation of `bit(x,j)`.

In figure 12.2, y is multiplied by a factor of 2 until $y > x$. A detail that needs to be taken into account is that y can overflow. The overflow can be detected by noting when y becomes negative.

`Math.sqrt()`: Since the calculation of $(y + 2^j)^2$ in figure 12.3 can overflow, the result may be an abnormally negative number. This problem can be addressed by (efficiently) changing the algorithm's *if* logic to

if $((y + 2^j)^2 \leq x)$ and $((y + 2^j)^2 > 0)$ then $y = y + 2^j$

12.3.2 String

As explained in section 12.1.4, string objects can be implemented as arrays. In a similar vein, all the string related services can be implemented as operations on arrays. An important implementation detail is that the actual length of the string must be maintained throughout these operations and that array entries beyond this length are not considered part of the string.

`String.intValue`, **`String.setInt`**: These functions can be implemented using the algorithms from figures 12.4 and 12.5, respectively. Note that both algo-

rithms don't handle negative numbers—a detail that must be handled by the implementation.

All other subroutines in this class are straightforward. Note that the ASCII codes of `newline`, `backspace`, and `doubleQuote` are 128, 129, and 34, respectively.

12.3.3 Array

Note that `Array.new()` is not a constructor, but rather a function (despite its name). Therefore, memory space for a new array should be explicitly allocated using a call to `Memory.alloc()`. Similarly, de-allocation of arrays must be done explicitly using `Memory.deAlloc()`.

12.3.4 Output

Character Bitmaps We suggest using character bitmaps of 11 rows by 8 columns, leading to 23 lines of 64 characters each. Since designing and building bitmaps for all the printable ASCII characters is quite a burden, we supply predefined bitmaps (except for one or two characters, left to you as an exercise). Specifically, we supply a skeletal `Output` class containing Jack code that defines, for each printable ASCII character, an array that holds its bitmap (implementing a font that we created). The array consists of 11 entries, each corresponding to a row of pixels. In particular, the value of entry j is a binary number whose bits represent the 8 pixels that render the character's image in the j-th row of its bitmap.

12.3.5 Screen

`Screen.drawPixel()`: Drawing a pixel on the screen is done by directly accessing the screen's memory map using `Memory.peek()` and `Memory.poke()`. Recall that the memory map of the screen on the Hack platform specifies that the pixel at column c and row r $(0 \leq c \leq 511, 0 \leq r \leq 255)$ is mapped to the $c\%16$ bit of memory location $16384 + r \cdot 32 + c/16$. Notice that drawing a single pixel requires changing a single bit in the accessed word, a task that can be achieved in Jack using bit-wise operations.

`Screen.drawLine()`: The algorithm from figure 12.8a can potentially lead to overflow. However, the efficiency improvement suggested in figure 12.8b also eliminates the overflow problem.

`Screen.drawCircle()`: Likewise, the algorithm from figure 12.9 can potentially lead to overflow. Limiting circle radii to be at most 181 avoids this problem.

12.3.6 Keyboard

In the Hack platform, the memory map of the keyboard is a single 16-bit word located at memory address 24576.

`Keyboard.keyPressed()`: This function provides "raw" (direct) access to this memory location and can be implemented easily using `Memory.peek()`.

`Keyboard.readChar`, **`Keyboard.readString`**: These functions provide "cooked" access to single character inputs and to string inputs, respectively. Proposed cooking instructions appear in figures 12.12 and 12.13.

12.3.7 Memory

`Memory.peek()`, **`Memory.poke()`**: These functions are supposed to provide direct access to the underlying memory. How can this be accomplished in a high-level language? As it turns out, the Jack language includes a trapdoor that enables programmers to gain complete control of the computer's memory. This hacking trick can be exploited to implement `peek` and `poke` using plain Jack programming.

The trick is based on an anomalous use of reference variables (pointers). Specifically, the Jack language does not prevent the programmer from assigning a constant to a reference variable. This constant can then be treated as an absolute memory address. In particular, when the reference variable happens to be an array, this trick can give convenient and direct access to the entire computer memory. Figure 12.4 gives the details.

Following the first two lines of figure 12.14, the base of the `memory` array points to the first address in the computer's RAM. To set or get the value of the RAM location whose physical address is `j`, all we have to do is manipulate the array entry

```
// To create a Jack-level "proxy" of the RAM:
var Array memory;
let memory = 0;
// From this point on we can use code like:
let x = memory[j]  // Where j is any RAM address
let memory[j] = y  // Where j is any RAM address
```

Figure 12.14 A trapdoor enabling complete control of the RAM from Jack.

`memory[j]`. This will cause the compiler to manipulate the RAM location whose address is `0+j`, which is precisely what is desired.

As we have pointed out earlier, Jack arrays are not allocated space on the heap at compile-time, but rather at run-time, when the array's `new` function is called. Here, however, a `new` initialization will defeat the purpose, since the whole idea is to anchor the array in a selected address rather then let the OS allocate it to an address in the heap that we don't control. In short, this hacking trick works because we use the array variable without allocating it "properly," as we would do in normal usage of arrays.

`Memory.alloc()`, **`Memory.deAlloc()`**: These functions can be implemented by either the basic algorithm from figure 12.6a on the improved algorithm from figure 12.6b using either *best-fit* or *first-fit*. Recall that the standard implementation of the VM over the Hack platform specifies that the heap resides at RAM locations 2048-16383.

12.3.8 Sys

`Sys.init`: An application program written in Jack is a set of classes. One class must be named `Main`, and this class must include a function named `main`. In order to start running the application program, the `Main.main()` function should be invoked. Now, it should be understood that the operating system is itself a program (set of classes). Thus, when the computer boots up, we want to start running the operating system program first, and then we want the OS to start running the main program.

With that in mind, the chain of command is implemented as follows. First, the VM (chapter 8) includes bootstrap code that automatically invokes a function called `Sys.init()`. This function, which is assumed to exist in the OS's `Sys` class, should then call all the `init()` functions of the other OS classes, and then call `Main.main()`. This latter function is assumed to exist in the application program.

`Sys.wait`: This function can be implemented pragmatically, under the limitations of the simulated Hack platform. In particular, you can use a loop that runs approximately *n* milliseconds before it (and the function) returns. 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, but that's life.

`Sys.halt`: This function can be implemented by entering an infinite loop.

12.4 Perspective

The software library presented in this chapter includes some basic services found in most operating systems, for example, managing memory, driving I/O, handling initialization, supplying mathematical functions not implemented in hardware, and implementing data types like the *string* abstraction. We have chosen to call this standard software library an "operating system" to reflect its main function: encapsulating the gory hardware details, omissions, and idiosyncrasies in a transparent software packaging, enabling other programs to use its services via a clean interface. However, the gap between what we have called here an OS and industrial-strength operating systems remains wide.

For starters, our OS lacks some of the very basic components most closely associated with operating systems. For example, our OS supports neither multi-threading nor multi-processing; in contrast, the very kernel of most operating systems is devoted to exactly that. Our OS has no mass storage devices; in contrast, the main data store kept and handled by operating systems is a file system abstraction. Our OS has neither a "command line" interface (as in a Unix shell or a DOS window) nor a graphical one (windows, mouse, icons, etc.); in contrast, this is the operating system aspect that users expect to see and interact with. Numerous other services commonly found in operating systems are not present in our OS, for example, security, communication, and more.

Another major difference lies in the interplay between the OS code and the user code. In most computers, the OS code is considered "privileged"—the hardware platform forbids the user code from performing various operations allowed exclusively to OS code. Consequently, access to operating system services requires a mechanism that is more elaborate than a simple function call. Further, programming languages usually wrap these OS services in regular functions or methods. 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 opera-

tions that we presented (e.g., line- and circle-drawing) are often also implemented in special graphics acceleration hardware.

Readers who wish to extend the OS functionality are welcome to do so, as we comment on in chapter 13.

12.5 Project

Objective Implement the operating system described in the chapter. Each of the OS classes can be implemented and unit-tested in isolation, and in any particular order.

Resources The main tool that you need for this project is Jack—the language in which you will develop the OS. Therefore, you also need the supplied Jack compiler to compile your OS implementation as well as the supplied test programs. In order to facilitate partial testing of the OS, you also need the complete compiled version of our OS, consisting of a collection of `.vm` files (one for each OS class). Finally, you need the supplied VM emulator. This program will be used as the platform on which the actual test takes place.

Contract Write a Jack OS implementation and test it using the programs and testing scenarios described here. Each test program uses a certain subset of OS services.

Testing Strategy

We suggest developing and unit-testing each OS class in isolation. This can be done by compiling the OS class that you write and then putting the resulting `.vm` file in a directory that contains the supplied `.vm` files of the rest of the OS. In particular, to develop, compile, and test each OS class `Xxx.jack` in isolation, we recommend following this routine:

1. Put, in the same directory, the following items: the OS class `Xxx.jack` that you are developing, all the supplied OS `.vm` files, and the relevant supplied test program (a collection of one or more `.jack` files).
2. Compile the directory using the supplied Jack compiler. This will result in compiling your `Xxx.jack` OS class as well as the class files of the test program. In the process, a new `Xxx.vm` file will be created, replacing the originally supplied OS class. That's exactly what we want: the directory now contains the executable test

program, the complete OS minus the original `Xxx.vm` OS class, plus your version of `Xxx.vm`.

3. Load the directory's code (OS + test program) into the VM emulator.
4. Execute the code and check that the OS services are working properly, according to the guidelines given below.

OS Classes and Test Programs

There are eight OS classes: `Memory`, `Array`, `Math`, `String`, `Output`, `Screen`, `Keyboard`, and `Sys`. For each OS class `Xxx` we supply a skeletal `Xxx.jack` class file with all the required subroutine signatures, a corresponding test class named `Main.jack`, and related test scripts.

Memory, Array, Math To test your implementation of every one of these OS classes, compile the relevant directory, execute the supplied test script on the VM emulator, and make sure that the comparison with the compare file ends successfully.

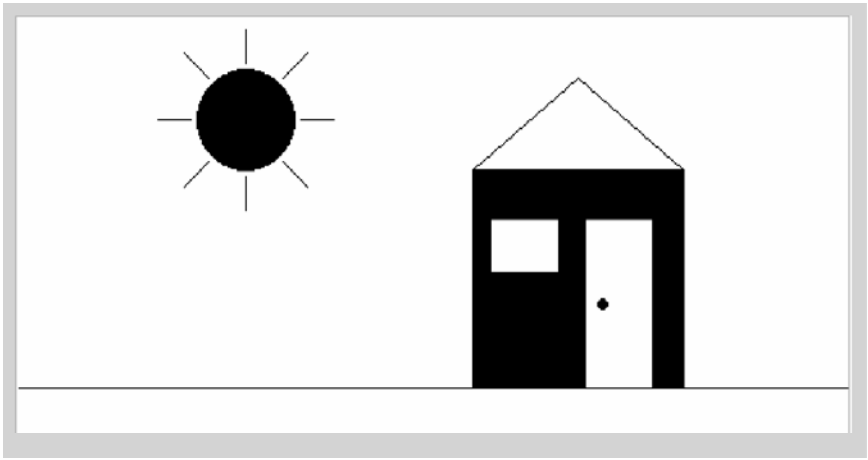
Note that the supplied test programs don't comprise a full test of the `Memory.alloc` and `Memory.deAlloc` functions. A complete test of these memory management functions requires inspecting internal implementation details not visible in user-level testing. Thus it is recommended that you test these two functions using step-by-step debugging in the VM emulator.

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

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



Keyboard This OS class is tested using a test program that effects some user-program interaction. For each function in the `Keyboard` class (`keyPressed`,

readChar, readLine, readInt), the program requests the user to press some keyboard keys. If the function is implemented correctly and the requested keys are pressed, the program prints the text "ok" and proceeds to test the next function. If not, the program repeats the request for the same function. If all requests end successfully, the program prints 'Test ended successfully', at which point the screen may look like this:

```
keyPressed test:
Please press the 'Page Down' key
ok
readChar test:
(Verify that the pressed character is echoed to the screen)
Please press the number '3': 3
ok
readLine test:
(Verify echo and usage of 'backspace')
Please type 'JACK' and press enter: JACK
ok
readInt test:
(Verify echo and usage of 'backspace')
Please type '-32123' and press enter: -32123
ok

Test completed successfully
```

Sys Only two functions in this class can be tested: Sys.init and Sys.wait. The supplied test program tests the Sys.wait function by requesting the user to press any key, then waiting for two seconds (using Sys.wait), and then printing another message on the screen. The time that elapses from the moment the key is released until the next message is printed should be 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 would work properly unless Sys.init is implemented correctly. A simple way to test Sys.init in isolation is to run the *Pong* game using your Sys.vm file.

Complete Test After testing successfully each OS class in isolation, test your entire OS implementation using the *Pong* game, whose source code is available in projects/12/Pong. Put all your OS .jack files in the Pong directory, compile the directory, and execute the game in the VM emulator. If the game works, then Mazel Tov! You are the proud owner of an operating system written entirely by you.