11. The Operating System[1]

"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 a modern, object-based Java-like 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 piece of software, which is missing in this puzzle, is an *operating system*. The OS is a large and complex program, designed to close gaps between the computer's software and hardware systems, and to make the overall computer more accessible to programmers and users. For example, our computer is equipped with a bitmap screen. In order to output the text "`Hello World`", several hundreds pixels must be drawn on specific locations on the computer's screen. To do so, we can consult the hardware specification, and write machine language commands that put the necessary bits in the RAM segment that controls the screen's output. Needless to say, programmers in high-level languages will need a better interface with the screen. They will want to use commands like `print('Hello World')`, and let someone else worry about the details. This someone else is the operating system.

The operating system is organized as a collection of software libraries, each handling a family of basic services. For example, text printing will be typically supported by a method like `printString(String s)`, which will typically be part of a class called `Output`. Thus, when the compiler will encounter a statement like `print('Hello World')` in some high-level code, it will translate it into a call to `Output.printString('Hello World')`. In a similar fashion, many mathematical operations, string processing functions, memory management routines, and so on, will be translated by the compiler into calls to respective OS routines, each dedicated to carry out one well-defined service.

In this chapter we specify and build a simple operating system, called Sack. The Sack OS features some fifty routines, organized in eight classes:

- `Math`: implements basic mathematical operations;
- `String`: implements the `String` type and basic string processing operations;
- `Array`: enables the construction and disposal of arrays;
- `Output`: handles text based output;
- `Screen`: handles graphic screen output;
- `Keyboard`: handles user input from the keyboard;
- `Memory`: handles memory operations;
- `Sys`: provides execution-related services.

[1] From *The Elements of Computing Systems*, Nisan & Schocken, MIT Press, forthcoming in 2003, www.idc.ac.il/csd

The chapter starts with an expanded background section that describes major algorithms and programming techniques used by operating systems in general and by the Sack OS in particular. Next, we specify the complete Sack API, and give guidelines on how to implement it in Jack.

Note the "symbiotic" relationship between Jack and Sack. On the one hand, Sack is written in Jack. On the other hand, Sack can be viewed as an extension of the Jack language. To give a historical perspective, we note in passing that the C language was originally invented in order to write the Unix operating system, which then served – among other things -- to extend the capabilities of C. This is similar to the Jack/Sack interplay.

The chapter embeds two key lessons, one in software engineering and one in computer science. First, we describe and illustrate the important interplay between high-level languages, compilers, and operating systems. Second, we present a series of elegant and efficient algorithms, each being a little computer science gem.

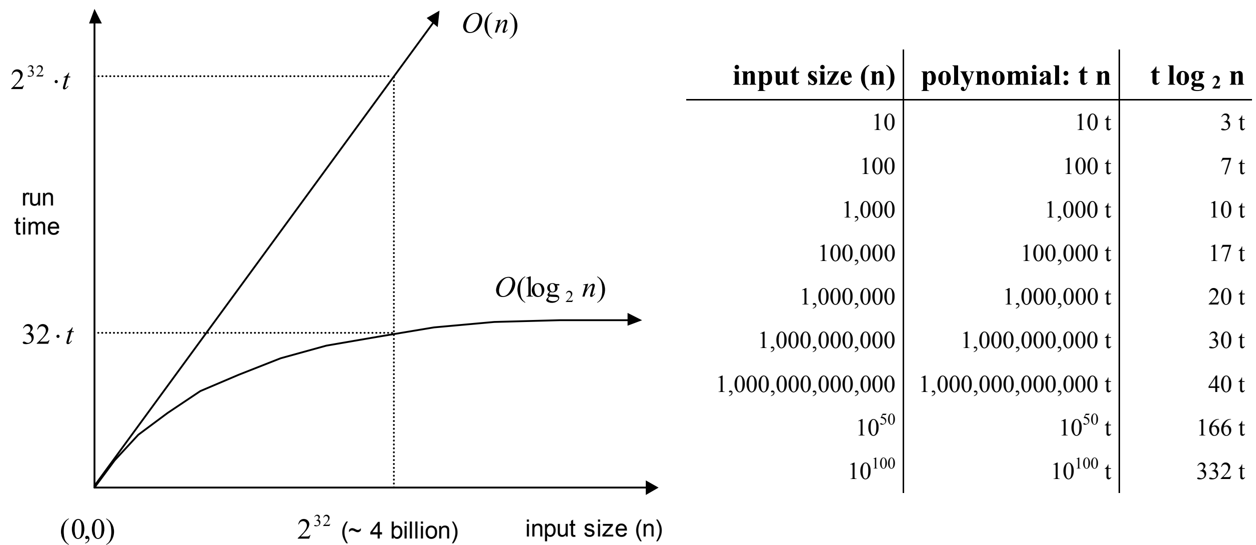
The chapter is work in progress.

11.1 Background

11.1.1 Efficiency First

The simplest way to multiply two numbers x and y is to add x to some register y times. To improve the performance of this naïve algorithm, we can first check which number is greater, and then add max(x,y) to the register, min(x,y) times. If we denote the smaller multiplicand n and the time it takes to add two numbers t, the run-time of this naïve multiplication algorithm will be $t \cdot n$. In computer science, such a run-time function is called "$O(n)$", which reads "an order of magnitude of n". This notation makes sense, since t varies greatly from one CPU to another, and thus is not a property of the algorithm. By leaving t out of the picture, the notation $O(n)$ implies that irrespective of the computer platform on which this algorithm is implemented, its run-time is *inherently* a polynomial function of the input size n.

When we write programs that solve real-life problems, nature often throws at us very large and very small numbers (which, in binary terms, have the same inflationary impact on the input size). Therefore, in many applications polynomial run-time is unacceptably slow. And that's why we always seek algorithms whose performance is insensitive to the input size. For example, consider a multiplication algorithm whose run-time is $(\log_2 n) \cdot t$ rather than $n \cdot t$. What would happen to the algorithm's performance when the input size is doubled from 1000 to 2000? While the polynomial run-time will slow down proportionally from $1000 \cdot t$ to $2000 \cdot t$, the logarithmic run-time will hardly change, from $10 \cdot t$ to $11 \cdot t$. Indeed, logarithmic run-time algorithms possess a magical property: the amount of work that they do is almost independent of the size of the underlying job. Fig. 11-1 provides an example of this remarkable phenomenon.



input size (n)	polynomial: t n	t log 2 n
10	10 t	3 t
100	100 t	7 t
1,000	1,000 t	10 t
100,000	100,000 t	17 t
1,000,000	1,000,000 t	20 t
1,000,000,000	1,000,000,000 t	30 t
1,000,000,000,000	1,000,000,000,000 t	40 t
10^{50}	10^{50} t	166 t
10^{100}	10^{100} t	332 t

FIGURE 11-1: Run time functions. The run-time of a typical algorithm is a function of *n*, the input size, and *t*, the time it takes to add two numbers on the underlying hardware. We see that polynomial run-time is dominated by *n* -- a variable that can assume very large values that we cannot control, whereas logarithmic run-time is dominated by *t* -- a predictable and tiny constant.

Efficient algorithms hold the key to improving the performance of the computer's operations; the more elementary and widely used the operation, the bigger will be the impact of a better algorithm. For example, later in the chapter we present a line drawing algorithm that involves many multiplication operations. Clearly, the time saving gained by using an efficient multiplication algorithm will have a dramatic impact on any graphical application that uses this line drawing algorithm.

We now turn to describe efficient algorithms for *multiplication*, *division*, and *square root* computations. The run-time of all these algorithms will be $O(\log_2 n)$, where *n* is the size of the algorithm's input.

11.1.2 Mathematical Operations

Almost every computer 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, e.g. multiplication and division, are implemented either in hardware or in software, depending on the computer's purpose and performance requirements. This section shows how multiplication, division, and square root operations can be implemented efficiently in software, at the operating system level.

Multiplication

Consider the standard multiplication method taught in elementary school. To compute 356 times 27, we write 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 numbers in the two rows and obtain the result. As we show below, the binary version of this algorithm is exactly the same (we use smaller numbers to save clutter):

the "steps"

		1	0	1	1	=	1	1
			1	0	1	=		5
		1	0	1	1			
	0	0	0	0				
1	0	1	1					
1	1	0	1	1	1	=	5	5

the algorithm explained (first 4 out of 16 iterations)

								bit(y,j)
0	0	0	1	0	1	1	x	
0	0	0	0	1	0	1	y	
0	0	0	**1**	**0**	**1**	**1**	shiftedx=x	**1**
0	0	**1**	**0**	**1**	**1**	0	shiftedx=2*shiftedx	**0**
0	**1**	**0**	**1**	**1**	0	0	shiftedx=2*shiftedx	**1**
1	**0**	**1**	**1**	0	0	0	shiftedx=2*shiftedx	0
0	1	1	0	1	1	1	sum where bit(y,j)=1	

```
// To multiply two n-bit numbers x and y:
initialize sum = 0
initialize shiftedx = x
for j = 0...(n-1) do {
    if bit(y,j) then sum += shiftedx
      shiftedx += shiftedx
    }
```

Algorithm 11-2: Binary multiplication of n-bit numbers (the example ignores some of the leading zeroes in each number). The right side of the example shows the first four iterations only, since in all the remaining iterations bit(y,j)=0, and thus the bottom line is unchanged.

What is the run-time of algorithm 11-2? If each iteration takes t time-units to complete, it is $t \cdot w$, where w is the word's width of the underlying computer. Since w represents the number of binary digits of the multiplied number n, we have $w = \log_2 n$. As it turns out, this $O(\log_2 n)$ algorithm is as good as we can possibly hope for. The only way to speed up the multiplication further is to implement it in hardware (a nice exercise -- the chip logic can be gleaned from the example).

Algorithm 11-2 should work as is for both unsigned and signed 2's complement numbers.

Implementation tip: Note that in each iteration j, the algorithm has to extract the jth bit of the second number. To save time, the OS could benefit from a general purpose `bit(x,j)` function, as follows:

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

There are various ways to implement this function efficiently, e.g. using a bit-wise `and` operator.

Division

One way to compute the division of x by y is to count how many times y can be subtracted from x before the remainder becomes less than x. To speed up this polynomial algorithm, we can inspect the magnitude of x and then subtract large chunks of y's 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. This is the rationale behind the well-known "school method" for division – the dreaded routine that many of us were trained to perform without understanding why it works. Algorithm 11-3 unveils the mystery.

the "steps"	the algorithm explained		
1 7 8	1 7 8	initialize remainder = 891	
8 9 1 | 5	8 9 1 | 5	how many 100's of 5 fit into the remainder?	100
5	5 0 0	remainder = remainder-100*5=391	
3 9	3 9 1	how many 10's of 5 fit into the remainder?	70
3 5	3 5 0	remainder = remainder-70*5=41	
4 1	4 1	how many 1's of 5 fit into the remainder?	8
4 0	4 0	**remainder** = remainder-8*5=1	
1	1	**result:**	178

```
// to divide (n-bit) x by y:
initialize remainder = x
initialize result = 0
for j = (n−1)...0 do {
    if remainder ≥ y·2^j then {
        result += 2^j
        remainder −= y·2^j
    }
}
```

Algorithm 11-3: Decimal division example (top) and Binary division algorithm (bottom).
The example and the algorithm are based on precisely the same strategy.

Note that the decimal and binary versions of the division algorithm are almost the same. The only difference is that instead of progressing in leaps of 10^j we progress in leaps of 2^j, $j = w-1, w-2, \ldots, 0$, where w is the number of digits of the divided number. Hence, the run-time of this algorithm is $O(\log_2 n)$, where n is the largest number that we can be asked to divide.

We end this section with several ways to improve this algorithm's performance.

Time efficiency improvement I: Note that in each iteration $j = (n-1)\ldots 0$, the algorithm has to compute 2^j. In programming terms:

`twoToThe(j)`: Returns the integer 2^j for $j = (n-1)\ldots 0$

This computation does not depend on the divided numbers. Thus we can compute it, once and for all, and store the values in an array `twoToThe[1..n]`. This can be done in the initialization routine of the operating system's `Math` library.

Time efficiency improvement II: Note that in each iteration $j = (n-1)\ldots 0$, the algorithm has to compute $y \cdot 2^j$. Thus it would make sense to factor this computation out of the loop, and begin the algorithm with something like the following code:

```
// compute all values of y·2^j, j = 0...(n-2), while trying to avoid overflow
yTimesTwoToThe[0]=y
for j=1...(n-2) do {
    yTimesTwoToThe[j]=2*yTimesTwoToThe[j-1]
     if yTimesTwoToThe[j] is overflowed then break
}
```

Space efficiency improvement: Consider the just-computed `yTimesTwoToThe[]` array. Unlike `twoToThe[]`, this array depends on one of the divided numbers. Thus we cannot pre-compute it outside the division method. At the same time, we can save memory space by implementing it as a *global array*. This way, it will not be re-allocated memory space in every invocation of the division function (of course we can recycle the array's space when the function returns, but here we run smack into time efficiency matters). Thus, it is recommended to create `yTimesTwoToThe[]` as a global array, and re-compute its entries each time the division method is entered.

Square Root

The square root function $y = \sqrt{x}$ has two convenient properties. First, it is monotonically increasing. Second, it's 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.

// To compute the integer part of $y = \sqrt{x}$:
find y such that $y^2 \leq x \leq (y+1)^2$
// recommended technique: binary search

Algorithm 11-4: Square root computation.

Implementation tip (for the Hack platform): Since the maximal positive number in the Hack/Jack platform is 2^{15} , the binary search should be done in the range [0...181].

11.1.3 Input/Output Management

Computers use a variety of display devices to communicate with users. The displayed images -- characters, pictures, and animation -- are rendered so quickly and effortlessly, that we tend to take their display mechanism for granted. In fact, computers have to work rather hard to draw anything on a screen -- even a single character. With that in mind, we now turn to describe what goes on behind the screen, so to speak. In particular, we focus on the key geometric algorithms used for drawing pixels, lines, and circles.

The physical screen can be viewed as a two-dimensional grid of *pixels* (shorthand for "pictorial elements"). For simplicity, let us assume a black-and-white display, in which the color status of each pixel can be represented by a single bit. How can we connect such a device to the computer? A common method is to take a certain memory segment, which we call "screen memory map", and use it to record the entire screen contents, one bit per pixel. To synchronize between the logical and physical representations of the screen contents, we introduce a process that continuously scans the screen memory map, and, for each bit, draws the respective pixel on the screen. Fig. 11-5 describes the overall process.

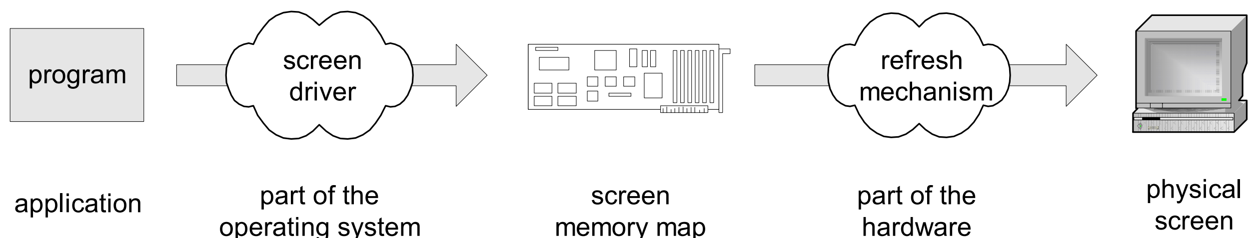


FIGURE 11-5: Screen management is essentially a two-stage process. The application "draws" the image in memory, via a screen driver interface. The actual image is then rendered on the screen by an infinite refresh loop. (The various simulators supplied with this book emulate the refresh logic using a software thread that runs parallel to the executing program).

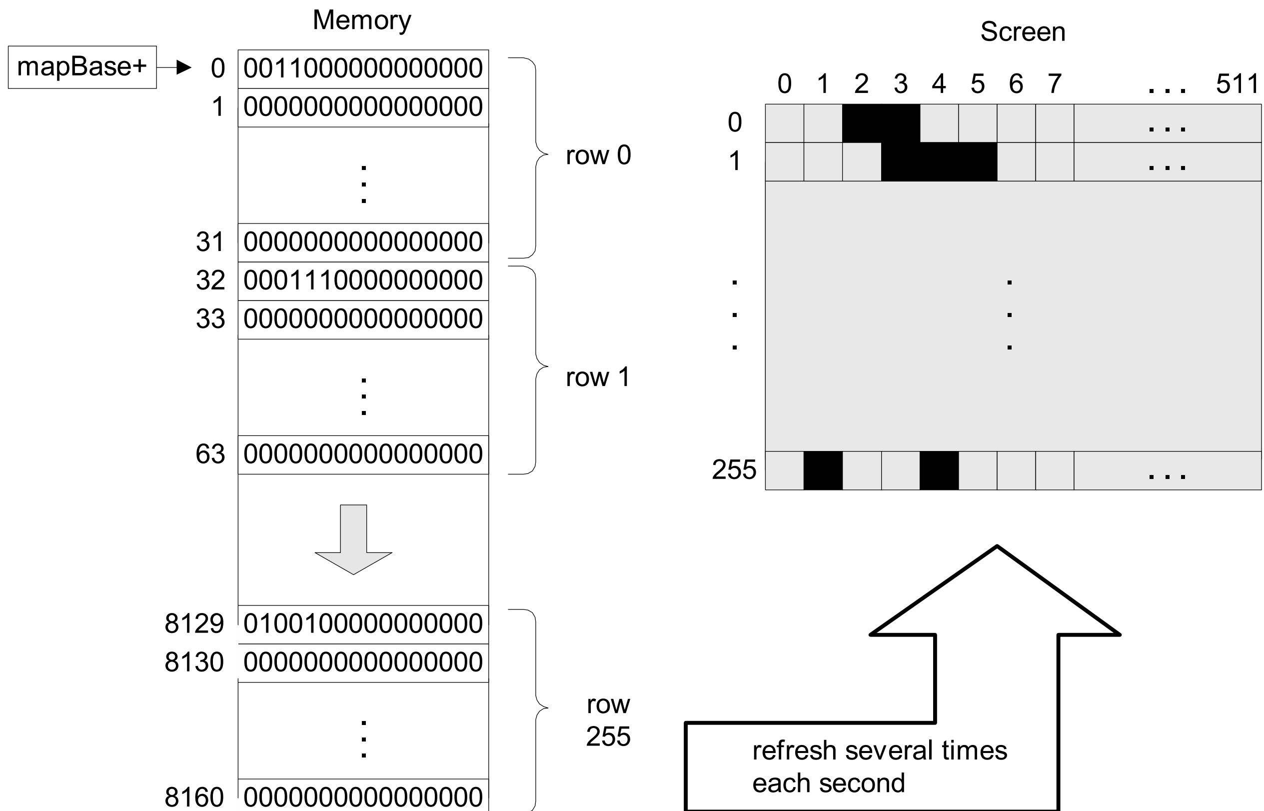
We see that from a software engineering perspective, the "brain" that regulates image drawing is the *screen driver*. This software module, which is part of the operating system, is a library of routines for drawing characters, lines, circles, etc. on the designated output device (each device

would normally have its own driver). Hence, when an application program wants to write or draw something on the screen, all it has to do is use commands like `print('hello world')` or `drawLine(50,75,212,317)`. The compiler will translate these high-level commands into calls to routines that will do the actual drawing. These routines will be organized as methods in the OS's screen driver library.

We now turn to describe algorithms for drawing pixels, lines, and circles. These primitives can be easily extended to the drawing of more complex geometric and free-form figures.

Pixel Drawing

The most elementary drawing operation performed by computers is that of drawing a single pixel. Pixels are specified using (*column*, *row*) coordinates. At the same time, the corresponding bits that represent the pixels in the computer's memory are arranged linearly. Thus, when a program requests the operating system to draw a pixel in screen location (*x*,*y*), the OS must figure out the address of the corresponding bit in the screen memory map. The general interface and mapping are described in Algorithm 11-6, using the Hack platform for illustration.



The general setting (in any computer)	**Hack Platform** (shown above)
screen size = ncols by *nrows*	screen size = 512 by 256
*npixels = ncols * nrows*	*npixels* = 512 * 256 = 131072
w = width of word in memory	*w* = 16
Memory map orientation: by rows or by cols.	Memory map orientation: by rows
blockSize = *ncols* / *w* or *nrows* / *w*	*blockSize* = 512 / 16 = 32
address(*x,y*)=*mapBase*+*blockSize***y*+*x*/*w*	address(*x,y*) = 16384+32**y*+*x*/16
bit(x,y) = in address(*x,y*), the *x*%*w bit*	**bit(x,y)**=in address(x,y), the *x*%16 *bit*

// `drawPixel(x,y)` : sets the pixel in screen location (x,y) to value:

`address = mapBase+blockSize*y+x/w`

use the `poke()` method to set the (x%w)-th bit in `address` to `value`

Algorithm 11-6: Pixel drawing.

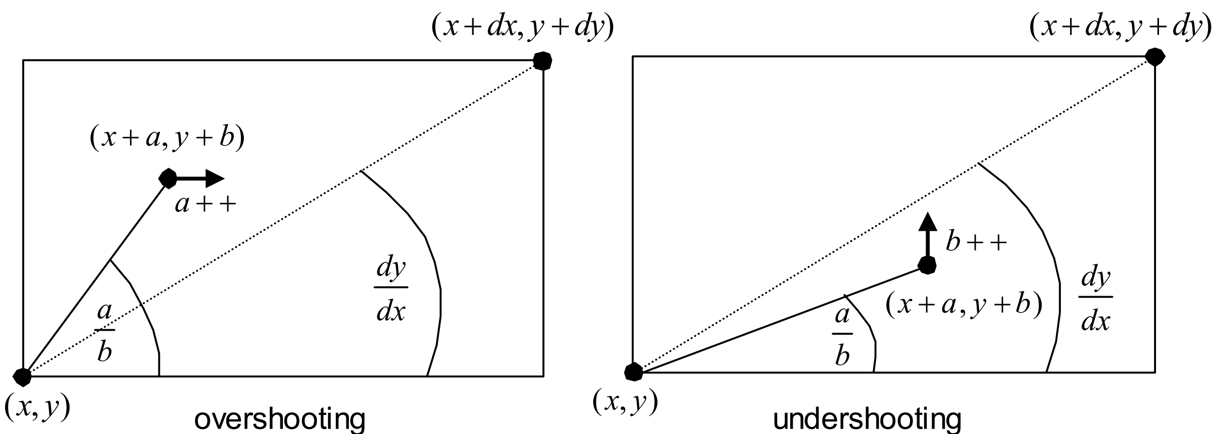
The `poke()` method mentioned in Algorithm 11-6 is also an operating system routine, described later in this chapter. Instead of invoking `poke()`, one can also access the memory location directly, using the same technique that `poke()` uses. This can yield a more efficient code.

Now that we know how to draw a pixel, we turn to describe how to draw a line.

Line Drawing

Most computers today use *raster*, also called *bitmap,* display technologies. The only operation that can be physically performed in such a screen is that of drawing a single pixel. Hence, when asked to draw a line between two screen locations, the best that we can possibly do is approximate the line by drawing a series of pixels along the imaginary line that connects 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 "biological pixels," there is a limit to the image granularity that the human eye can resolve. Thus, high-resolution screens and printers can fool the human eye to believe that the lines drawn by pixels or printed dots are actually 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 the (x2,y2) pixel, until it is reached. See Algorithm 11-7 for the details.



// To draw a line between points (x, y) and $(x + dx, y + dy)$
// (assuming $dx, dy \geq 0$)
initialize $(a, b) = (0,0)$
while $a \leq dx$ or $b \leq dy$ do {
    drawPixel$(x + a, y + b)$
    if $a \cdot dy \prec b \cdot dx$ then a++ else b++
}

Algorithm 11-7: Line Drawing

Algorithm 11-7 is applicable only for $dx, dy \geq 0$. To extend it into a general-purpose line drawing routine, one also has to take care of the three other possibilities: $dx, dy \prec 0$, $dx \succ 0, dy \prec 0$, and $dx \prec 0, dy \succ 0$.

One problem with Algorithm 11-7 is that the multiplication operation can cause overflows. We present another line-drawing technique, called "Midpoint Algorithm" or "Bresenham Algorithm", that uses integers only and no multiplications.

// To draw a line between points (x_0, y_0) and (x_1, y_1)
// assuming $x_1 \succ x_0$ and $dy \leq dx$, i.e. $0 \leq slope \leq 1$.

```
drawLine(x0, y0, x1, y1) {
   x = x0                // current pixel location, initialized with (x0,y0)
   y = y0
   dx = x1 – x0          // horizontal and vertical distances
   dy = y1 – y0
   d = 2 * dy – dx       // initialize the decision variable
   deltaE = 2 * dy       // decision variable increment for east
   deltaNE = 2 * (dy – dx)  // decision variable increment for north-east
   drawPixel(x,y)
   while (x < x1) {
     if (d < 0) {        // move east
        d += deltaE
        x++
     }
     else {              // move north-east
        d += deltaNE
        x++, y++
     }
     drawPixel(x,y)
   }
}
```

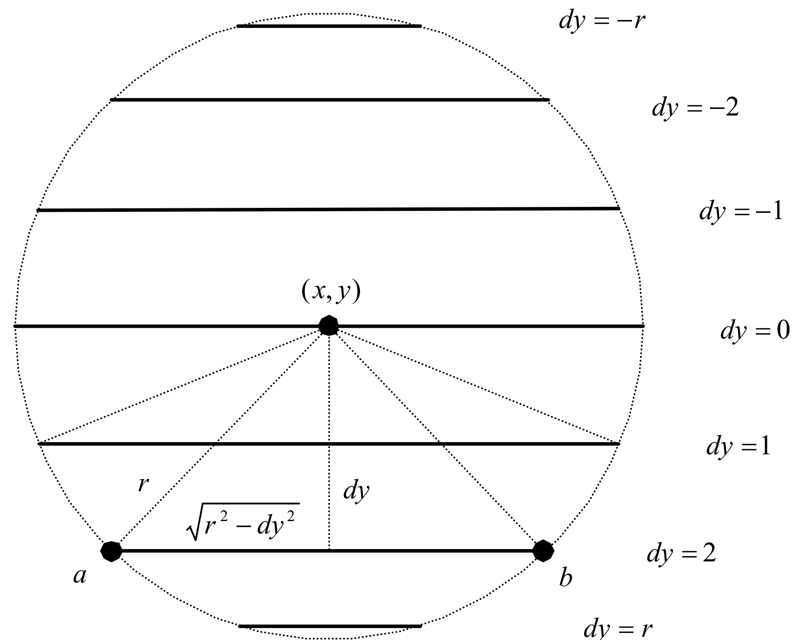
Algorithm 11-8: "Midpoint Algorithm" for line drawing

This algorithm is very similar to Algorithm 11-7, as it also utilizes the dy / dx ratio. For each x in the range $x_0 \leq x \leq x_1$, a decision is made whether to move east (right) or north east (right and up). The decision is made according to a "decision variable" that is updated in each iteration according to the last decision made (the rationale is based on the implicit form of a line, $f(x, y) = ax + by + c = 0$).

Algorithm 11-8 assumes that $x_1 \succ x_0$ and $dy \leq dx$, i.e. $0 \leq slope \leq 1$. The algorithm should be extended to support different line orientations. The first assumption can be handled by swapping the source and destination points when needed. The second can be handled by inverting the X and Y axes such that the slope will be in the correct range.

Circle Drawing

There are several ways to draw a circle on the screen. We present an algorithm that uses three routines already implemented in this chapter: *multiplication*, *square root* computation, and *line drawing*.



point $a = (x - \sqrt{r^2 - dy^2}, y + dy)$ point $b = (x + \sqrt{r^2 - dy^2}, y + dy)$

// To draw a filled circle of radius r around point (x, y):

for each $dy \in -r \ldots r$ do

drawLine from $(x - \sqrt{r^2 - dy^2}, y + dy)$ to $(x + \sqrt{r^2 - dy^2}, y + dy)$

Algorithm 11-9: Circle Drawing

The algorithm is based on drawing a series 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 screen row along the circle's north-south diameter, and thus the resulting circle is completely filled. We avoid overflow in the computation of r^2 by placing a limit on *r*'s magnitude; see the `drawCircle` function API for details.

Keyboard Handling

The interface between the computer and the keyboard can be implemented by a RAM location, called *keyboard memory map*. Since Unicode is 16-bit, the map can be implemented as a single 16-bit word in memory. When a key is pressed on the keyboard, the hardware is responsible for setting the value of the keyboard map to the Unicode (or some other unique code) of the pressed key; when no key is pressed, a code like 0 can be stored.

The OS routine that handles the keyboard is typically called `keyPressed()`. When called, this routine returns the current value of the keyboard memory map. The implementation of the `keyPressed()` routine of the Sack OS can be easily done by consulting the keyboard mapping in the Hack hardware specification.

11.1.4 Array Management

Modern operating systems should enable high level languages to create and manipulate variable size arrays. For example, consider a Jack program designed to implement some multi-player game:

```
var Array players;
let players=Array.new(Keyboard.readInt('how many players?'));
...
players.dispose;
```

(`readInt(String msg)` is yet another OS method that displays `msg` on the screen and returns the user's input as an integer).

In the Java language (and unlike Pascal), arrays are created in two stages. At *compile-time*, the `var` statement only creates a pointer variable whose name is set to that of the declared array. The array itself is constructed and allocated memory at *run-time*, when the OS's `Array.new` routine is called. As the example illustrates, this flexibility allows us to allocate only as much memory space as necessary.

In order to support this dynamic array creation protocol, the OS programmer can implement arrays as object instances of a class called `Array`. Importantly, the `Array` class should not have a constructor, since we cannot know at compile-time how much memory to allocate to each array instance. Instead, the OS programmer can define an `Array.new(int size)` function that, when called, allocates to the array as much heap space as implied by the `size` argument. Similarly, one should define an `Array.dispose` method that, when called, de-allocates the memory space presently occupied by the current array object.

11.1.5 String Processing

A modern OS should support variable-length strings. This can be done by creating a `String` class that provides the string abstraction and related services. The standard data structure used in this context is typically designed to hold an array of characters that holds the string contents, the maximum length of the string, and the current length of the string.

The Sack OS has a `String` class with routines for creating instances of this data structure and implementing all the functionality described in the `String` library of the OS API. These operations must have O(1) run-time.

11.1.6 Memory Management

Memory Access

Operating systems carry out low-level memory access via two methods traditionally called *peek* and *poke*. The `peek(address)` method gets the integer which is the value of the given RAM address. The `poke(address,value)` method sets the value of the given RAM location to the given value. In order to implement these methods, the language in which the OS is written must provide means for accessing specific RAM addresses directly. Different high level languages employ different tricks to facilitate such direct access to the hardware, while some languages attempt to prevent the programmer from doing so.

A Jack implementation: The Jack language includes a trapdoor that enables the programmer to gain complete control of the computer's memory. 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.

```
// To create a Jack-level "proxy" of the RAM:
var Array memory;
let memory=0;
```

From this point on, 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 `memory[j]`. This will cause the compiler to manipulate the RAM location whose address is `0+j`, which is precisely what we want to do.

Recall that in Jack, arrays are not allocated space on the heap at compile-time, but rather at run-time, when the array's `new` method is called. Here, however, a `new` initialization will defeat the purpose, since the whole idea is to anchor the array in a particular 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 initializing it "properly", as we would do in normal usage of arrays.

Memory Allocation

Programs create objects. These objects must be stored and managed in memory -- a task normally handled by the operating system. In particular, when a running program constructs a new object of a certain size, enough RAM space must be located in memory and then allocated to store the new object. When the program declares that the object is no longer needed, its RAM space may be recycled. In the computer architecture jargon, the RAM segment which is designated to storing objects is called *heap.*

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 memory allocation 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*, that points to the beginning of the yet un-allocated part of the heap. Algorithm 11-10 gives the details.

```
// objects are stored on the heap.
Initialization: free=heapBase

// to allocate size words in memory:
function alloc(size):
   pointer = free
   free += size
   return pointer

// to de-allocate the memory space of a given object:
function deAlloc(object):
   do nothing
```

Algorithm 11-10: Basic Memory Allocation Scheme (wasteful)

Algorithm 11-10 is wasteful, as it does not reclaim the space of decommissioned objects.

Improved memory allocation algorithm: This algorithm manages a linked list of available memory blocks, called *freeList*. Each block is characterized by two “housekeeping” fields: the block’s length, and a pointer to the next block in the *freeList*. These fields can be kept in the two memory locations preceding the block itself, (i.e. `b.length==x[-1]`, and `b.next==x[-2])`.

When asked to allocate a memory block of size *n*, the algorithm has to search the *freeList* for a suitable block. Two well known options for doing this are called *best-fit* and *first-fit*. *Best-fit* finds the block whose size is the closest (from above) to the required size, while *first-fit* finds the first block that is long enough. Once the block has been found, the required memory segment is taken from it. Next, the allocated block is updated in the *freeList,* becoming to be the part that remained after the allocation (if no memory was left in the block, it is eliminated from the *freeList*).

When asked to reclaim the memory of an unused object, the algorithm inserts the de-allocated block into the *freeList*. The details are given in Algorithm 11-11.

```
// objects are stored on the heap.
Initialization:
   freeList = heapBase+2
   freeList.length = heapEnd-(heapBase+2)
   freeList.next = null

// to allocate size words in memory:
function alloc(size):
   1. use methods like best-fit or first-fit
      to locate a free block in freeList
   2. return the base address of that block

// to de-allocate the memory space of a given object:
function deAlloc(object):
   Append the object to the freeList
```

Algorithm 11-11: Improved Memory Allocation Scheme (with memory recycling)

Dynamic memory allocation schemes like Algorithm 11-11 may create a block fragmentation problem. Hence, some kind of "defrag" operation should be considered, i.e. merging memory segments that are physically consecutive in memory but logically split into different blocks in the *freeList*. The defragmentation operation can be done each time an object is de-allocated, or when alloc() cannot find an appropriate block, or according to some intermediate or ad-hoc condition.

11.1.7 Booting

An application program written in Jack is a collection of classes. One class must be named `main`, and this class must include a method named `main`. In order to start running the application program, the `Main.main` method should be invoked. Now, it should be understood that the operating system is itself a program. When the computer boots up, we want to start running the operating system program first, and then we want the OS to tell the application program to start running as well. This is accomplished by an OS function called `Sys.init()`, which is automatically invoked by the VM's bootstrap code (see section 7.3.1). The `Sys.init` function should call all the `init()` methods of all the OS classes (libraries), and then call the `Main.main()` method of the application program.

11.2 The Sack OS Specification

Sack is a basic operating system featuring a set of basic services, implemented as a library of subroutines. Compilers of high level languages over the Hack platform assume that these subroutines exist, and invoke them in the code that they generate.

Sack is organized as a collection of classes that can be used by application programs. These classes run in normal user mode and are not different from the classes of the application. The Sack subroutines are grouped into the following classes:

- **Math:** Provides basic mathematical operations;
- **String:** Implements the `String` type and basic string-related operations;
- **Array:** Enables the construction and disposal of arrays;
- **Output:** Handles text based output;
- **Screen:** Handles graphic screen output;
- **Keyboard:** Handles user input from the keyboard;
- **Memory:** Handles memory operations;
- **Sys:** Provides some execution-related services.

In terms of implementation, the Sack API is partitioned into 4 levels. *Level-0* subroutines are needed for any compiler; *Level-1* subroutines provide very basic services; *Level-2* subroutines are needed for programs of reasonable functionality; *Level-3* includes all the remaining subroutines. Each Sack implementation must specify the level that it supports.

Math

This class enables various mathematical operations. The implementation level of each function is listed at the end of the function's description.

- Function void **`init`**(). *Level-0.*
- Function int **`abs`**(int x): Returns the absolute value of x. *Level-1.*
- Function int **`multiply`**(int x, int y): Returns the product of x and y. *Level-0.*
- Function int **`divide`**(int x, int y): Returns the integer part of the x/y. *Level-0.*
- Function int **`min`**(int x, int y): Returns the minimum of x and y. *Level-1.*
- Function int **`max`**(int x, int y): Returns the maximum of x and y. *Level-1.*
- Function int **`sqrt`**(int x): Returns the integer part of the square root of x. *Level-2.*

String

This class implements the `String` data type and various string-related operations. The implementation level of each function is listed at the end of the function's description.

- Function void **`init`**(). *Level-0.*

- Constructor String **`new`**(int maxLength): Constructs a new empty string (of length zero) that can contain at most maxLength characters. *Level-0*.
- Method void **`dispose`**(): Disposes this string. *Level-0*.
- Method int **`length`**(): Returns the length of this string. *Level-1*.
- Method char **`charAt`**(int j): Returns the character at location j of this string. *Level-1*.
- Method void **`setCharAt`**(int j, char c): Sets the j'th element of this string to c. *Level-1*.
- Method String **`appendChar`**(char c): Appends c to this string and returns this string. *Level-0*.
- Method void **`eraseLastChar`**(): Erases the last character from this string. *Level-1*.
- Method int **`intValue`**(): Returns the integer value of this string (or at least the prefix part, i.e. until a non numeric character is found). *Level-3*.
- Method void **`setInt`**(int j): Sets this string to hold a representation of j. *Level-3*.
- Function char **`backSpace`**(): Returns the backspace character. *Level-1*.
- Function char **`doubleQuote`**(): Returns the double quote (") character. *Level-1*.
- Function char **`newLine`**(): Returns the newline character. *Level-1*.

Array

This class enables the construction and disposal of arrays. Both routines in this class are *level-0*.

- Function Array **`new`**(int size): Constructs a new array of the given size.
- Method void **`dispose`**(): Disposes this array.

Output

This class allows writing text on the screen. All the functions in this class except for `init()` are *level-3*.

- Function void **`init`**(). *Level-0.*
- Function void **`moveCursor`**(int i, int j): Moves the cursor to the j'th column of the i'th row, and erases the character located 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.

Screen

This class allows drawing graphics on the screen. It accesses the screen directly, using the screen's memory-mapped address as defined by the hardware. All the functions in this class are *level-2*.

- Function void **`init`**()
- Function void **`clearScreen`**(): Erases the entire screen.
- Function void **`setColor`**(boolean b): Sets the screen 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 where the top left corner is (x1, y1) and the bottom right corner is (x2,y2).
- Function void **`drawCircle`**(int x, int y, int r): Draws a filled circle of radius r around (x,y). The radius *r* must be 181 or less.

Keyboard

This class allows reading inputs from the keyboard.

- Function void **`init`**(). *Level-0.*
- Function char **`keyPressed`**(): Returns the character of the currently pressed key on the keyboard; if no key is currently pressed, returns 0. *Level-1*.
- Function char **`readChar`**(): Waits until a key is pressed on the keyboard and released, and then echoes the key to the screen and returns the character of the pressed key. *Level-3*.
- Function String **`readLine`**(String message): Prints the message on the screen, reads the next line (until a newline character) from the keyboard, echoes the line to the screen, and returns its value. This method handles user backspaces. *Level-3*.
- Function int **`readInt`**(String message): Prints the message on the screen, reads the next line (until a newline character) from the keyboard, echoes the line to the screen, and returns the integer until the first non numeric character in the line. This method handles user backspaces. *Level-3*.

Memory

This class allows direct access to the main memory. All the functions in this class are *level-0,* except for `peek()` and `poke()`, which are *level-1*

- Function void **`init`**().
- Function int **`peek`**(int address): Returns the value of the main memory at this address.
- Function void **`poke`**(int address, int value): Sets the value of the main memory at this address to the given value.

- Function Array **`alloc`**(int size): Allocates the specified space on the heap and returns a reference to it.
- Function void **`deAlloc`**(Array o): De-allocates the given object and frees its memory space.

Sys

This class supports some execution-related services. All the functions in this class are *level-1,* except for `init()`, which is *level-0*.

- Function void **`init`**(): Makes all initializations required and then calls the users' Main.main() method.
- 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. Implementation note: this can be implemented by a loop that runs approximately *n* milliseconds before it (and the method) returns. You will have to time your specific computer to obtain a one millisecond wait (this constant varies from one CPU to another). As a result, your `Sys.wait` function will not be portable, but that's life.

11.3 Perspective

Work in progress.

11.4 Build It

Sack is a simple operating system, similar to early versions of DOS.

Objective: Develop the Sack "kernel", which includes key parts of its `Memory`, `Math`, `Screen`, `Array`, `String`, and `System` libraries.

Contract: Implement (in the Jack programming language) the *level-0*, *level-1*, and *level-2* services of the operating system libraries, as defined in the Sack OS API (Section 11.2). Test your OS implementation by compiling and running the test programs supplied below.

Guidelines

The source code of the Sack operating system is written in Jack (just like Unix is written in C). The executable OS is a collection of `.vm` files (just like Windows is a collection of `.exe` and `.dll` files). The OS routines are implemented as VM functions, stored in these files. One of these functions is `Sys.init()`, stored in `Sys.vm`.

In a similar fashion, a Jack application is also a collection of `.vm` files (just like Word is a collection of `.exe` and `.dll` files). One of these files must be `Main.vm`., and this file must contain a VM function called `Main.main`. This function is invoked automatically by `Sys.int()` when the computer is reset.

Thus, a natural way to test a newly developed OS routine is to compile and execute an application that uses this routine. For example, to test the OS's multiplication method, we can write a Jack program that includes commands like "`let b=5; let x=b*12;`". If the OS is implemented properly, this code should put the number 60 in the memory location that was allocated for x. Hence, to complete the testing, we can compile both the OS and the application, run the resulting (overall) code, and inspect the memory.

Incremental testing: The following strategy can help you manage your tests. Recall that an executable version of the OS is already available in the book software suite. This OS is a collection of several VM files, one for each OS library (e.g. `Memory.vm`, `Screen.vm`, etc., each being the compiled version of corresponding `Memory.jack`, `Screen.jack`, etc. classes -- these are the classes that *you* have to develop and test in this project).

Now, in each part of the present project you have to develop routines in one of these libraries. For example, let us assume that you are now working on the `Memory.jack` class. To help "localize" your testing, you can put all the supplied OS `.vm` files in your working directory, and then replace the supplied `Memory.vm` file with your own version of this library, i.e. the `Memory.vm` file that *you* obtained by compiling *your* version of `Memory.jack`. This strategy makes sense, since the test programs that we supply below call other OS routines that you have not yet developed, and if these routines will not be present in the working directory you will get run-time errors.

Implementing the OS level-0 and level-1 services

This part of the project deals with implementing all the OS routines labeled as *level-0* and *level-1* in the Sack OS API (Section 11.2). The test program consists of three classes:

- **`Person.jack:`** this class represents a person with a name and an age. Contains a variety of text manipulation operations on the person's name.
- **`List.jack:`** this class creates an array and executes various multiplication and division operations on its entries.
- **`Main.jack:`** this class creates a person object and a list object and then tests various OS methods.

Guidelines: Implement all the *level-0* and *level-1* routines of the operating system. Use the Jack compiler to compile your OS `.jack` files. The result will be a set of "OS `.vm` files." Compile the supplied test program (3 classes above). The result will be a set of "application `.vm` files." Put both sets of `.vm` files in the same directory. Use the VM Emulator to execute the resulting code. Important: before you do so, disable the emulator's animation.

If the OS was implemented correctly, RAM locations 10000…10008 should contain the following values:

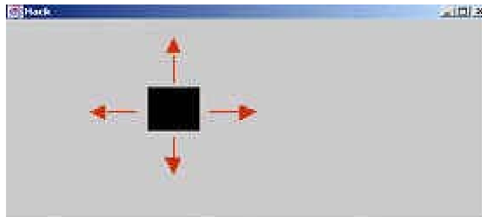
Address	Value
10000	1
10001	20
10002	2
10003	68
10004	76
10005	65
10006	66
10007	69
10008	3

Implementing the OS level-2 services

This part of the project deals with implementing all the OS routines labeled as *level-0* and *level-1* in the Sack OS API (Section 11.2). The test program is a simple game consisting of three classes:

- **`Square.jack:`** this class represents a graphical square. Contains a variety of graphical operations that use the OS screen drawing services.
- **`SquareGame.jack:`** this class starts a "game" in which a square is created and manipulated by the user.
- **`Main.jack:`** this class starts a new game.

Guidelines: Same as in "*Implementing the OS level-0 and level-1*" section. If the OS was implemented correctly, a square should appear on the screen, as follows:



The user should then be able to control the square movement by pressing the following keyboard keys:

right arrow:	move the square to the right;
left arrow:	move the square to the left;
up arrow:	move the square up;
down arrow:	move the square down;
x:	increment the square size by 2 pixels;
z:	decrement the square size by 2 pixels;
q:	quit the game.

To control the animation speed, you can change the delay constant in the `moveSquare` method in `SquareGame` class.