COMP 2718: The OS, Shell, Terminal, and Text

By: Dr. Andrew Vardy

Adapted from the notes of Dr. Rod Byrne

Outline

- What is an Operating System?
- The Shell and Terminal
- How are Characters Encoded?
- Standard Input, Standard Output, and Standard Error
- Escape Sequences
    - Example Programs

What is an Operating System?

The **operating system** (OS) on your computer manages the system's resources (processor, memory, files, . . .) and provides interfaces to the user and to user programs.

The following are the most common OSs:

- Linux
- Mac OS X
- Android
- Windows
- iOS

The top three have a common heritage as they are all variants of the Unix operating system and inherit the Unix command line (not obvious in Android).

The Shell

Modern OSs are layered to provide consistent interfaces between layers and ultimately to the computer's hardware.

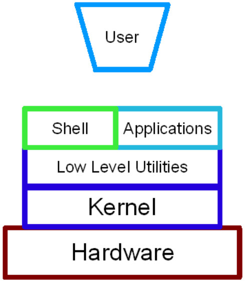


Figure 1: The Linux OS: penguintutor.com

This is for the Linux OS. The **shell** is a special application designed to let users interact with the OS in the same way that applications do. The shell provides the CLI.

The Shell and Terminal

Modern shells are based on the Unix shell. There are a number of different shells available, but we will focus on the most popular for Linux and Mac OS X: `bash` (Bourne Again SHell)

A **terminal emulator** (often just **terminal**) provides a graphical interface to interact with the shell. We refer to them as terminal *emulators* because the original terminals were just keyboard/display devices connected to a remote computer...

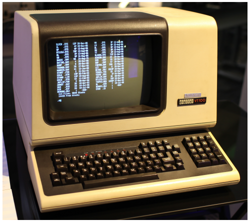


Figure 2: The DEC VT100 terminal: wikipedia.com

Text In, Text Out

A terminal displays a grid of characters. There are two streams of characters, one coming out for display, and one coming in from the keyboard.

The standard terminal size is 24 rows by 80 columns. Terminals can be re-sized arbitrarily, but it is sometimes nice to limit line length to 80 characters to respect this standard.

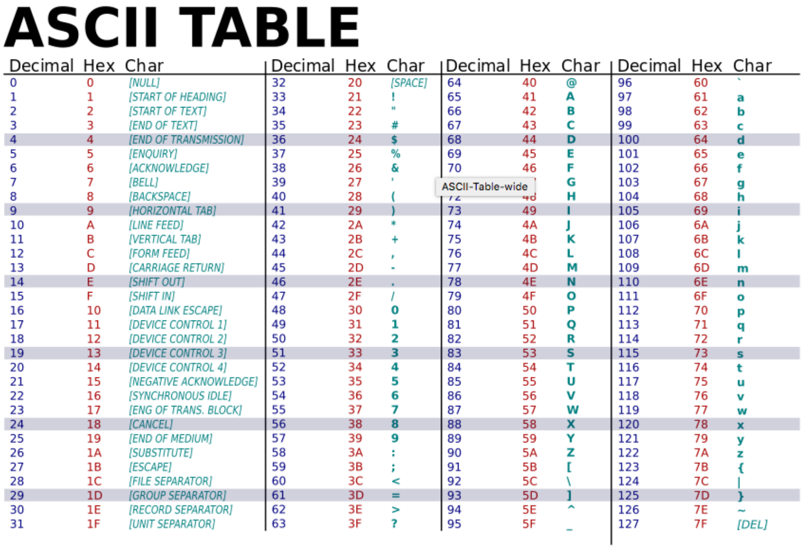
The **cursor** is a highlighted place in the terminal's grid, marking where the next typed character will be placed.

How are Characters Encoded?

There are many ways to map characters from the broad array of human languages into binary numbers. The following are among the most common:

- ASCII – 7 bit code – 128 characters
- ISO-8859-1 – 8 bit code for Western European languages
- Unicode – characters for most human languages
- UTF-8 – 8 bit encoding of Unicode

ASCII is a relatively simple scheme that remains useful to understand for computing. It includes the 26 latin letters in both cases, punctuation, as well as a set of control characters that are not intended to be printed but control how text is processed. For example, code 8 represents a backspace.



ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	[illegible]	[illegible]	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Figure 3: wikipedia.com

Standard Input, Standard Output, and Standard Error

Individual programs interact with the shell and each other through streams of characters called **stdin**, **stdout**, and **stderr** which have the following defaults:

- **standard input** (stdin) is the input that comes from the keyboard
- **standard output** (stdout) is the program's output displayed by the terminal
- **standard error** (stderr) are the program's error/debug messages displayed by the terminal

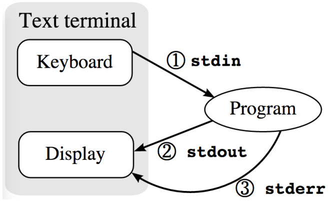


Figure 4: wikipedia.com

Writing to **stdout** in Java

The following Java program writes the characters `hi!` to stdout:

```java
public class H {
    public static void main(String [] args) {
        java.lang.System.out.println("hi!");
    }
}
```

Compile by executing the command line `javac H.java`. Execute with `java H`.

Writing to both **stdout** and **stderr** in Java

This program is the same as the previous except that it also outputs to standard error.

```
public class H2 {
    public static void main(String [] args) {
        java.lang.System.err.println("START");
        java.lang.System.out.println("hi!");
        java.lang.System.err.println("END");
    }
}
```

Upon running this you will see no real difference between stdout and stderr. However, we can use **redirection** (introduced later on) to redirect stderr to a file, so that the program's output is separated from debugging/error messages.

```
java H2 2> H2.txt
```

>2 means redirect standard error (stream 2) to the following file. (More on this later).

Reading from **stdin** and writing to **stdout** in Python

The following Python program reads lines of characters from stdin and prints them in uppercase to stdout:

```
import sys
while True:
    l = sys.stdin.readline()
    if len(l) == 0: break
    print l.upper()
```

There is no need to compile as Python is an interpreted language (more later on what is meant by ‘compile’ and ‘interpreted’). Execute with `python upper.py`. By default, the input comes from the keyboard so you will need to type to see results. Terminate with ‘control-D’ (End-Of-File in ASCII).

Escape sequences

An **escape sequence** is a sequence of characters that together represent a non-standard character or action. Some escape sequences just move the cursor:

- `\t` moves the cursor horizontally by one ‘tab’
- `\n` moves to the next line
- `\r` moves back to the start of the row
- `\b` erases the last character

Notice how the \ character allows the meaning of the next character to “escape” out of its normal interpretation. If you want a literal backslash then use `\\`.

There are many other escape sequences. My favourite is `\a` which causes the terminal to emit a beep!

```
echo -e "Oops\a"
```

Writing to **stdout** in C using **escape sequences**

```c
#include <stdio.h>
int main() {
    /* Draw 80 '*' chars. */
    for (int i=0; i<80; i++)
        printf("*");
    printf("\n");

    /* Draw '*' chars at each tab position. */
    for (int i=0; i<9; i++)
        printf("\t*");
    printf("\n");
    return 0;
}
```

We can even do some primitive animation by incorporating sleep instructions within the code:

```c
#include <stdio.h>
#include <unistd.h>
int main() {
    /* Without this line the terminal would wait
       for a newline before printing anything. */
    setbuf(stdout, NULL);

    /* Draw 80 '*' chars, pausing after each one. */
    for (int i=0; i<80; i++) {
        printf("*");
        usleep(20000);
    }
    printf("\n");
    return 0;
}
```

On the next couple of slides there are a couple of variations that make use of the following escape sequences:

- `\r` (return to the beginning of the line)
- `\b` (backspace)

These next two programs will run indefinitely. Terminate them with ‘control-c’ from the terminal.

progress_bar2: Draw 80 asterisks, erase them all, then start again.

```c
#include <stdio.h>
#include <unistd.h>
int main() {
    setbuf(stdout, NULL); /* Needed as before. */
    while (1 == 1) {
        for (int i=0; i<80; i++) {
            printf("*");
            usleep(20000);
        }
        /* Back to start, erase, then back to start. */
        printf("\r");
        for (int i=0; i<80; i++)
            printf(" ");
        printf("\r");
    }
    return 0; /* We'll never actually get here. */
}
```

progress_bar3: Draw 80 asterisks, erase one-by-one, then start again.

```c
#include <stdio.h>
#include <unistd.h>
int main() {
    setbuf(stdout, NULL); /* Needed as before. */
    while (1 == 1) {
        for (int i=0; i<80; i++) {
            printf("*");
            usleep(20000);
        }
        /* Backspace to the beginning of the line. */
        for (int i=0; i<80; i++) {
            printf("\b \b");
            usleep(20000);
        }
    }
    return 0; /* We'll never actually get here. */
}
```