COMP1521 25T2

Week 8 Lecture 1

File systems and Text Encoding and Unicode

Adapted from Angela Finlayson, Dylan Brotherston's,
Andrew Taylor and John Shepherd's slides

Today's Lecture

- File systems
  - Recap
  - Useful file system functions
- Representing Text
  - ASCII
  - Unicode
    - UTF8 encoding

Recap Exercise

Question 1: Assume I have a opened 2 files for writing and have FILE * f1 and f2 variables.

- What would the following write to the files? Would they depend on the systems I ran them on?

```
uint16_t x = 0xABCD;
fwrite(&x, 2, 1, f1);

uint8_t low_byte = x & 0xFF;
uint8_t high_byte = (x >> 8);
fputc(low_byte, f2);
fputc(high_byte, f2);
```

Recap Exercise

Question 2:

How would this be represented in octal:

rw-r - - - - -

Question 3:

If I ran the following on the command line:

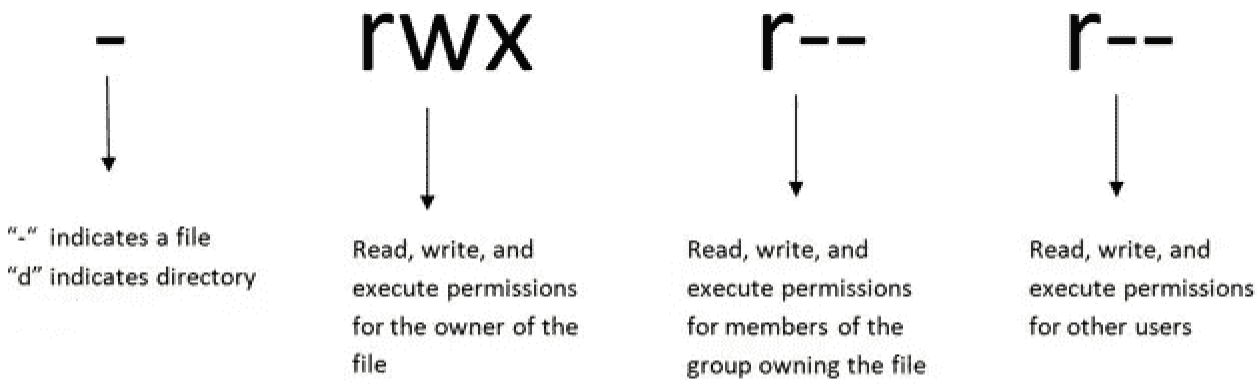
```
chmod 755 f
```

A. Would “others” have the execute permission set for the file f?
B. How could I check this from my C code?

File Permissions

Every file and directory in linux has read, write and execute permissions (access rights) for each of the following user groups:

- user: the file’s owner
- group: the members of the file's group
- other: everyone else
- type `ls -l` on command line to see



- rwx r-- r--

"-" indicates a file
"d" indicates directory

Read, write, and execute permissions for the owner of the file

Read, write, and execute permissions for members of the group owning the file

Read, write, and execute permissions for other users

C library wrapper for stat system call

```
int stat(const char *pathname, struct stat *statbuf);
```

- returns metadata associated with **pathname** in **statbuf**
- metadata returned includes:
    - inode number
    - type (file, directory, symbolic link, device)
    - size of file in bytes (if it is a file)
    - permissions (read, write, execute)
    - times of last access/modification/status-change

- returns **-1** and sets **errno** if metadata not accessible

C library wrapper for stat system call

```
int lstat(const char *pathname, struct stat *statbuf);
```

- same as stat() but doesn't follow symbolic links
    - in other words gives you metadata about the symbolic link and not the file it links to
    - important not to get stuck in infinite loops

```
int fstat(int fd, struct stat *statbuf);
```

- same as stat() but gets data via an open file descriptor

See **man 2 stat**

man 3 stat

man 7 inode

definition of struct stat

man 3 stat

```
struct stat {
  dev_t     st_dev;      /* ID of device containing file */
  ino_t     st_ino;      /* Inode number */
  mode_t    st_mode;     /* File type and mode */
  nlink_t   st_nlink;    /* Number of hard links */
  uid_t     st_uid;      /* User ID of owner */
  gid_t     st_gid;      /* Group ID of owner */
  dev_t     st_rdev;     /* Device ID (if special file) */
  off_t     st_size;     /* Total size, in bytes */
  ...
};
```

st_mode field of struct stat

man 7 inode

`st_mode` is a bitwise-or of these values (& others):

S_IFLNK	0120000	symbolic link
S_IFREG	0100000	regular file
S_IFDIR	0040000	directory
S_IRUSR	0000400	owner has read permission
S_IWUSR	0000200	owner has write permission
S_IXUSR	0000100	owner has execute permission
S_IRGRP	0000040	group has read permission
S_IWGRP	0000020	group has write permission
S_IXGRP	0000010	group has execute permission
S_IROTH	0000004	others have read permission
S_IWOTH	0000002	others have write permission
S_IXOTH	0000001	others have execute permission

Making a directory

```
int mkdir(const char *pathname, mode_t mode);
```

returns 0 if successful, returns -1 and sets **errno** otherwise

- for example: `mkdir("newDir", 0755)`

if **pathname** is e.g. \`a/b/c/d\`

- all of the directories \`a\`, \`b\` and \`c\` must exist
- directory \`c\` must be writable to the caller
- directory \`d\` must not already exist

the new directory contains two initial entries

- \`.\` is a reference to itself
- \`..\` is a reference to its parent directory

Demo: mkdir.c

Opening and Reading directories

```c
// open a directory stream for directory name
DIR *opendir(const char *name);

// return a pointer to next directory entry
struct dirent *readdir(DIR *dirp);

// close a directory stream
int closedir(DIR *dirp);
```

Found in man 3

Demo list_directory.c

Useful Linux (POSIX) functions

```
chmod(char *pathname, mode_t mode) // change permission of file/...
unlink(char *pathname) // remove a file...
rename(char *oldpath, char *newpath) // rename a file/directory
chdir(char *path) // change current working directory
getcwd(char *buf, size_t size) // get current working directory
link(char *oldpath, char *newpath) // create hard link to a file
symlink(char *target, char *linkpath) // create a symbolic link
```

Demo: chmod.c rm.c rename.c my_cd.c getcwd.c nest_directories.c many_links.c chain_links.c

Home Directory

~ means home directory in Linux

To get this value we can use

```
char *getenv(const char *name);
```

Example:

```
printf("%s", getenv("HOME"));
```

Text Representation

How should we represent text?

- We know how to represent unsigned integers, signed integers and real values in C.
- Text is arguably the most important data type
  - It can represent all other data types via serialization
    - E.g. JSON, XML, YAML, etc…
- Text == sequences of characters
- So how can we represent characters?

So, how should we represent characters?

- By default in C and MIPS we have used ASCII
- Modern computers use something called “UNICODE” to represent the individual characters!
- But other things came before…

A timeline of character representations

- 1828: First electronic Telegraph system (Pavel Schilling)
- 1837: Cooke and Wheatstone Telegraph
- 1844: Morse Code
- 1897: First radio transmission

many other encoding schemes that we won't cover

- 1943: First (modern) computer (Colossus)
- 1963: **ASCII**
- 1970s: **Extended ASCII**
- 1963: EBCDIC
- 1987: **Unicode**

Disclaimer:

- Note: this timeline is very **Western-centric**.
  - There are many other encoding schemes from around the world
- East Asian languages have particularly interesting ones
  - Due to writing systems with very large character sets
  - Some interesting examples include
    - (1980) The Chinese Character Code for Information Interchange
    - (1980) The GB 2312 standard
    - (1984) The Big5 Encodings
    - (1990s) Windows code pages 874 (Thai), 932 (Japan), 936 (Chinese)...

ASCII: 1963

- American Standard Code for Information Interchange
    - created by the American Standards Association (ASA)
    - later became the American National Standards Institute (ANSI)
        - the first organization to standardize the C programming language
- 7-bit (fixed-size) encoding
    - 128 possible values
    - all of the values are used
- One of the most common and influential encodings in computing

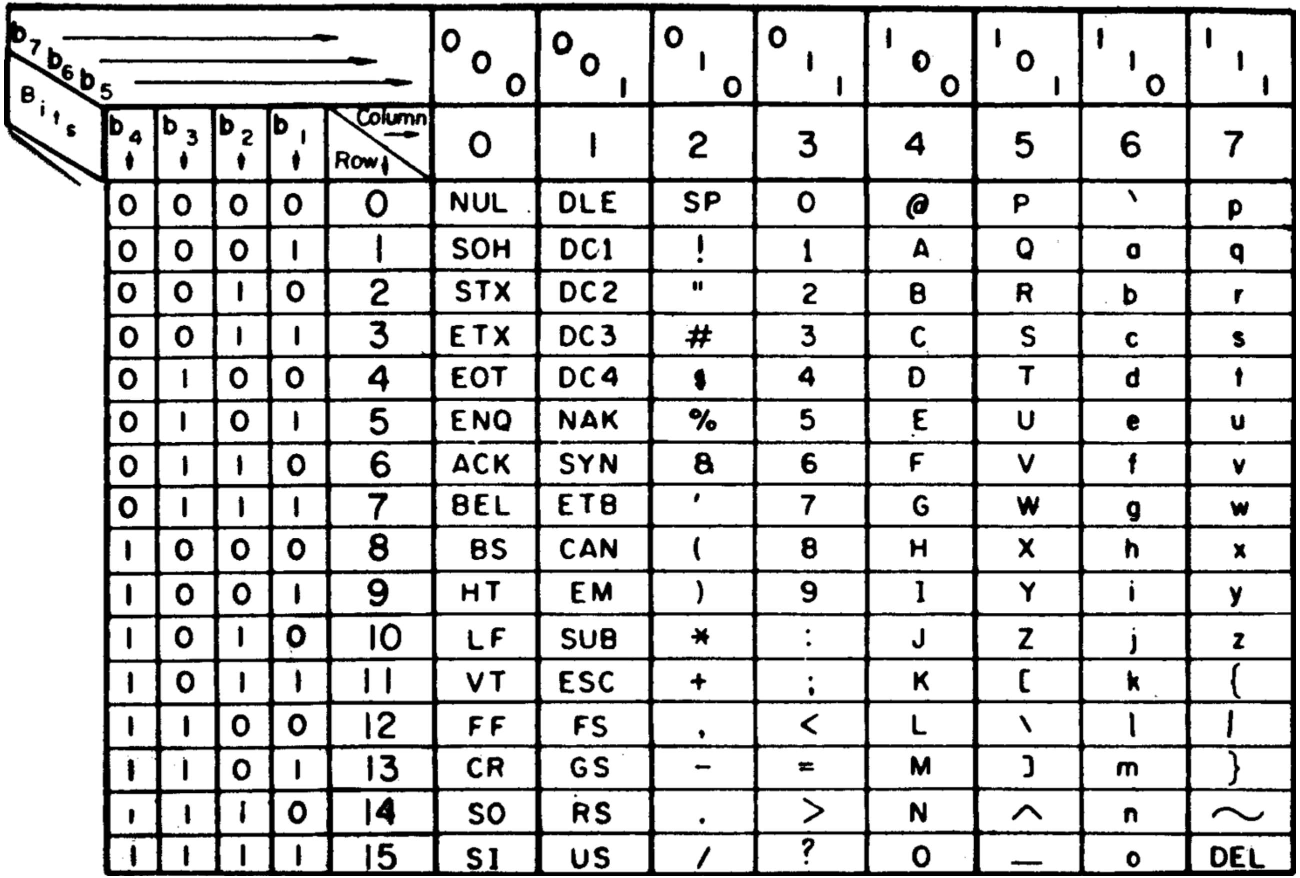
ASCII: Control Characters

- When ASCII was created, computers didn’t use monitors.
- Instead, they used teletypes — electromechanical devices with a keyboard for input and a printer for output.
  - These could be controlled by a human (typing) or by a computer (printing).
- Because the output was a physical mechanism, ASCII included control characters to
  - move the "carriage"
  - start a new line
  - ring the bell

ASCII: TTY (TeleTYpewriter)

ASCII:

USASCII code chart



b_7 b_6 b_5 Bits					0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b_4	b_3	b_2	b_1	Column → Row ↓	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	-	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

ASCII Overview

- Uses values in the range `0x00` to `0x7F` (0..127)
- Characters partitioned into sequential blocks (sticks)
    - control characters (sticks 0 and 1) (codes 0x00 to 0x1F)
        - e.g. '\0', '\n'
    - Punctuation (stick 2, parts of sticks 3..7)
    - digits (stick 3) (codes 0x30-0x39)
        - e.g. '0'..'9'
- upper case alphabetic (65..90) \... 'A’..'Z'
- lower case alphabetic (97..122) \... 'a’..'z'

ASCII Patterns

- Sequential nature of groups allows for helpful things like
  - Converting character digits into integers
    - ‘4’ - ‘0’ gives us the integer 4
  - Iterating through the alphabet, comparing letters
    - ‘a’ + 1 gives us ‘b’ and also ‘a’ < ‘b’
  - Case conversion
    - ‘A’ + 32 gives us ‘a’
  - Some patterns are not so helpful…
    - ‘<’ + 2 gives ‘>’
    - ‘[‘ + 2 gives ‘]’
    - ‘{‘ + 2 gives ‘}’
    - ‘(‘ + 2 gives ‘*’

ASCII: Bit Patterns

- The digits have values of 0b011 followed by the digits binary value
  - Allows for fast conversion between ASCII and binary numbers
- Uppercase and Lowercase letters are placed such that:
  - the only difference between them is the 5th bit
  - this allows for very fast case conversion and case insensitive string comparison

ASCII Demo

- ASCII_to_DEC.c
  - Convert from ascii character digit to a numeric decimal digit
- ASCII_case_insensitive.c
  - Convert to and from upper case and lower case characters

ASCII Limitations

- ASCII works well for English (American English)
- And is fairly decent for British English.
    - Unless you use the pound sign (£)
- But it doesn’t work well for other european languages
    - and doesn’t work at all for other languages (like Asian languages).
- The solution (for other European languages at least) was to use the 8th bit to extend the encoding.

Extended ASCII

EASCII is not standardized! So there are many different encodings

- All legitimate “Extended ASCII”
- KOI-8: Russian encoding
- ISO 8859-1 (aka Latin-1): Western European encoding
- Code page 899: DOS mathematical
- symbols etc…

(wikipedia lists 100s of different Code Pages)

This made EASCII perfect for *mojibake* disasters

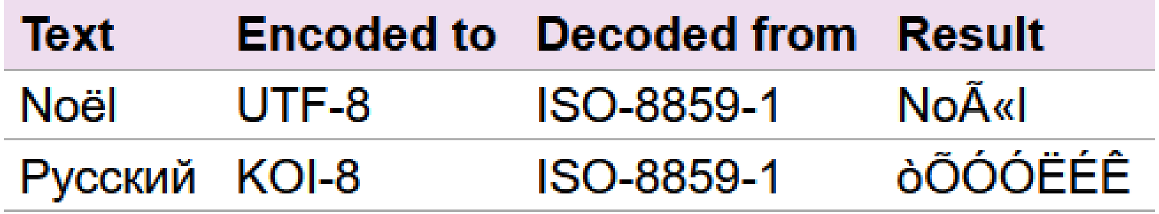
Mojibake

Mojibake occurs when:

- a byte string is decoded using the **wrong character encoding**, or
- two byte strings encoded in **different encodings** are concatenated

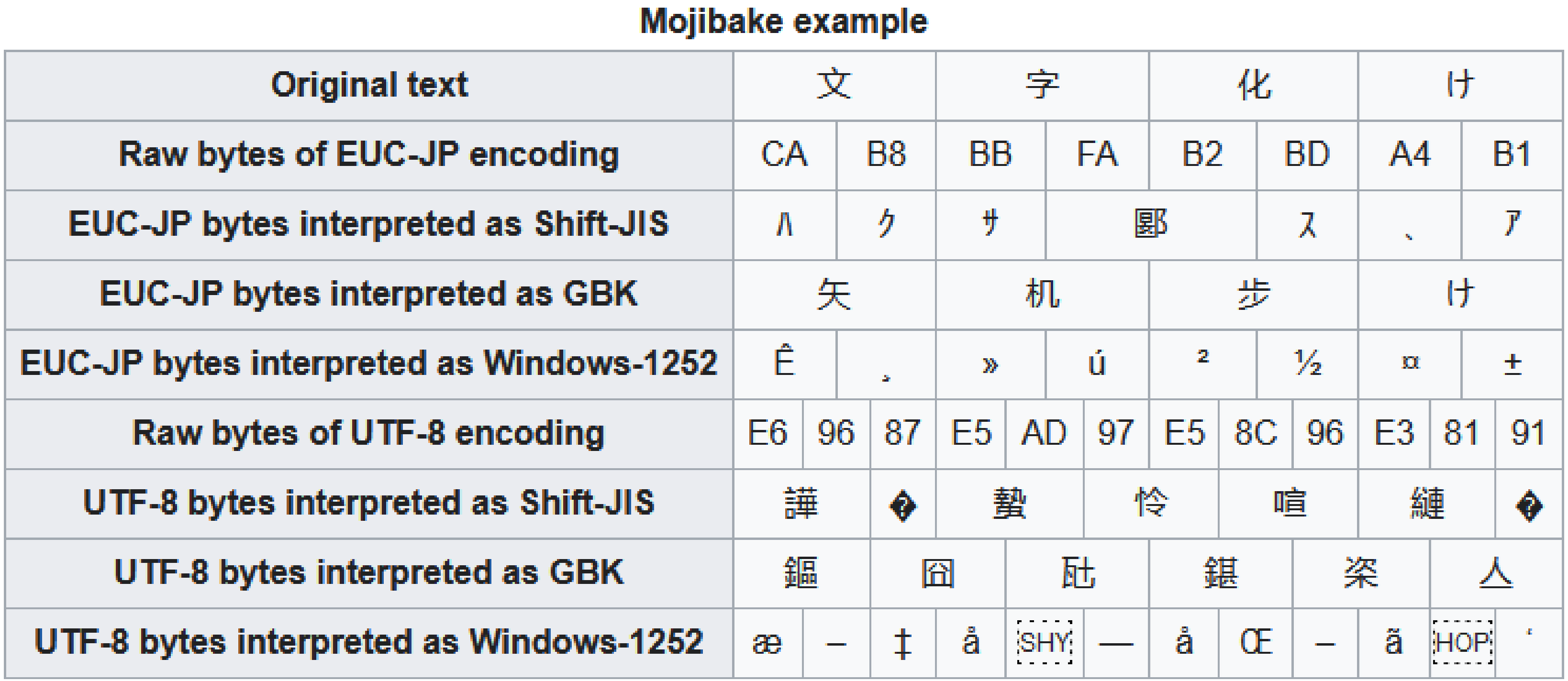
This results in garbled, unreadable characters

Examples:



Text	Encoded to	Decoded from	Result
Noël	UTF-8	ISO-8859-1	NoÃ«l
Русский	KOI-8	ISO-8859-1	òÕÓÓËÉÊ

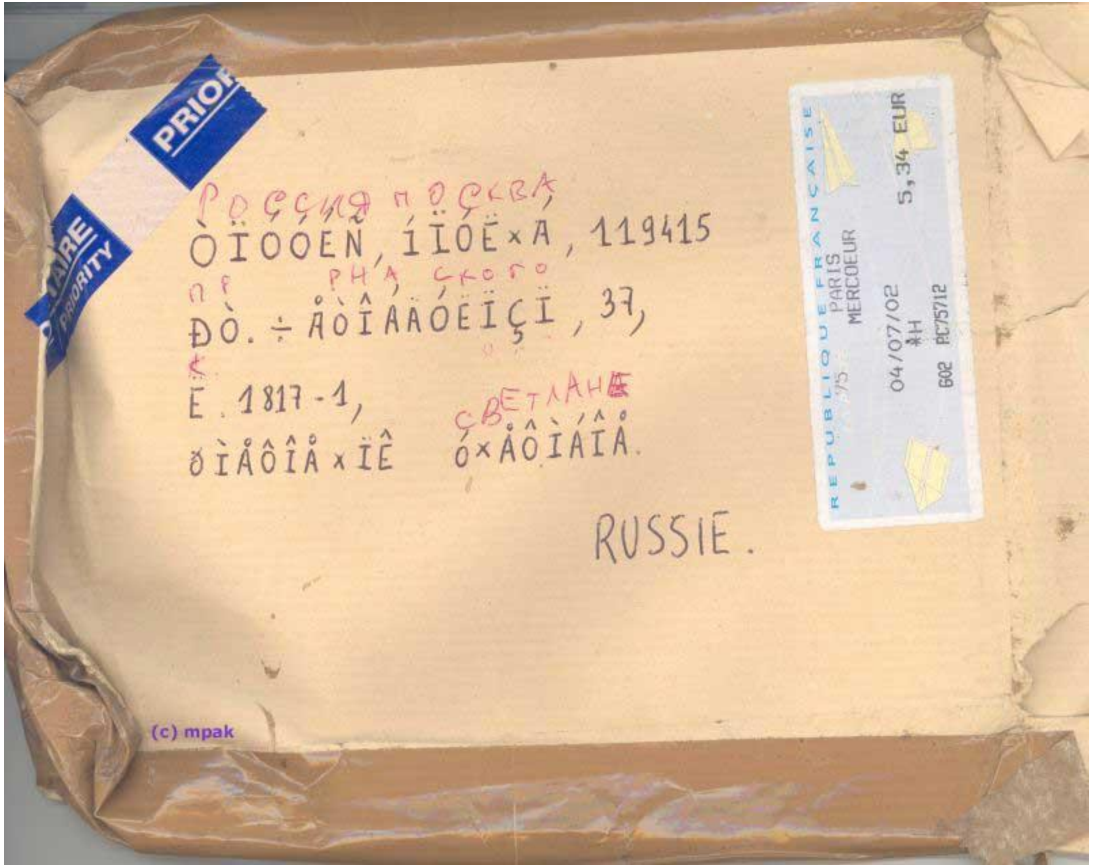
Mojibake (cont.)

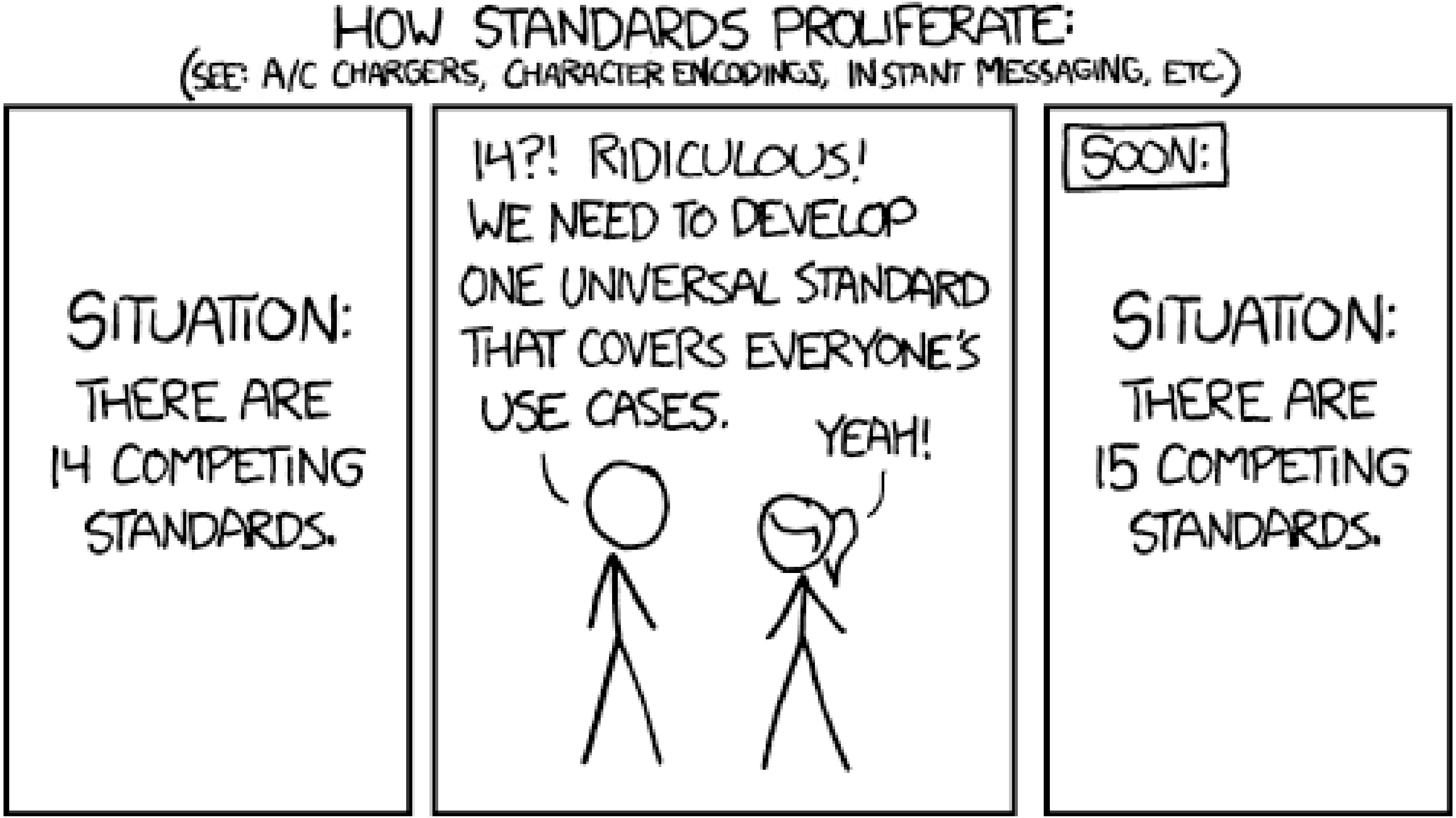


Mojibake example

<table>
<tr><th>Original text</th><td colspan="4">文</td><td colspan="4">字</td><td colspan="4">化</td><td colspan="4">け</td></tr>
<tr><th>Raw bytes of EUC-JP encoding</th><td colspan="2">CA</td><td colspan="2">B8</td><td colspan="2">BB</td><td colspan="2">FA</td><td colspan="2">B2</td><td colspan="2">BD</td><td colspan="2">A4</td><td colspan="2">B1</td></tr>
<tr><th>EUC-JP bytes interpreted as Shift-JIS</th><td colspan="2">ﾊ</td><td colspan="2">ｸ</td><td colspan="2">ｻ</td><td colspan="4">鄧</td><td colspan="2">ｽ</td><td colspan="2">､</td><td colspan="2">ｱ</td></tr>
<tr><th>EUC-JP bytes interpreted as GBK</th><td colspan="4">矢</td><td colspan="4">机</td><td colspan="4">步</td><td colspan="4">け</td></tr>
<tr><th>EUC-JP bytes interpreted as Windows-1252</th><td colspan="2">Ê</td><td colspan="2">¸</td><td colspan="2">»</td><td colspan="2">ú</td><td colspan="2">²</td><td colspan="2">½</td><td colspan="2">¤</td><td colspan="2">±</td></tr>
<tr><th>Raw bytes of UTF-8 encoding</th><td>E6</td><td colspan="2">96</td><td>87</td><td>E5</td><td colspan="2">AD</td><td>97</td><td>E5</td><td colspan="2">8C</td><td>96</td><td>E3</td><td colspan="2">81</td><td>91</td></tr>
<tr><th>UTF-8 bytes interpreted as Shift-JIS</th><td colspan="3">譁</td><td>�</td><td colspan="3">蟄</td><td colspan="2">怜</td><td colspan="3">喧</td><td colspan="3">縺</td><td>�</td></tr>
<tr><th>UTF-8 bytes interpreted as GBK</th><td colspan="3">鏂</td><td colspan="2">囧</td><td colspan="3">瓧</td><td colspan="3">鍖</td><td colspan="2">栬</td><td colspan="3">亼</td></tr>
<tr><th>UTF-8 bytes interpreted as Windows-1252</th><td>æ</td><td colspan="2">–</td><td>‡</td><td>å</td><td colspan="2">SHY</td><td>—</td><td>å</td><td colspan="2">Œ</td><td>–</td><td>ã</td><td colspan="2">HOP</td><td>‘</td></tr>
</table>

Mojibake IRL




HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC)
SITUATION: THERE ARE 14 COMPETING STANDARDS.
14?! RIDICULOUS! WE NEED TO DEVELOP ONE UNIVERSAL STANDARD THAT COVERS EVERYONE'S USE CASES.
YEAH!
SOON:
SITUATION: THERE ARE 15 COMPETING STANDARDS.

UNICODE

- UNICODE is maintained by the Unicode Consortium
- The goal of UNICODE is to create a single encoding that can represent all of the characters in all of the languages in the world.
- There are currently 149,878 characters in UNICODE.
- https://en.wikipedia.org/wiki/List_of_Unicode_characters

UNICODE: Codespace

- UNICODE is so large and has a very structured layout to try and make it more intuitive
- The Unicode Standard defines a codespace, (ie “The encoding”)
    - The Unicode codespace ranges from 0x0000 to 0x10FFFF
    - Each hex value represents a code point (i.e. a character)
- This gives a total of 1,114,112 code points
    - (293,168 are currently assigned) - approximately 25%.

UNICODE: Layout

These 1.1 million code points are split into 17 planes

- Plane 0 - 0x0000 - 0xFFFF
  - the Basic Multilingual Plane (BMP)
  - the vast majority of characters for most modern languages
- Plane 1 mostly contains historical characters and notation
  - Hieroglyphs e.g. 𓀀 𓀁 𓀂 𓀃
  - musical symbols e.g. 𝄠 𝄡 𝄢 𝄣
  - Emoji e.g. 😃 😇 😨 😴
- Plane 2 contains mainly additional Chinese, Japanese and Korean (CJK) characters

UNICODE: Layout

- Plane 3 is mostly unused but contains additional CJK characters
- Planes 4 - 13 are unassigned planes
- Plane 14 is the Supplementary Special-purpose Plane (SSP)
- Plane 15 -16 are set aside for private usage

Storing UNICODE characters: UTF-32

- The code points range from 0x0000 to 0x10FFFF
    - So we need at least 21 bits to represent them.
- We can use 32 bits to represent a single character.
- UTF-32 is a fixed width encoding
    - Simply take the UNICODE code point and store it in 32 bits.

UTF-32: Example

A → U+0041 → 0b00000000000000000000000001000001
€ → U+20AC → 0b00000000000000000010000010101100
字 → U+5B57 → 0b00000000000000000101101101010111
😀 → U+1F600 → 0b00000000000000011111011000000000

U+XXXX is the representation of a raw UNICODE code point

- code points are always at least 4 hex digits.
- 4 digit code points are on the 0th plane
- The 5th digit (if there is one) is the plane number

UTF-32: is very very inefficient

- Representing the largest code point, U+10FFFF would waste 11 bits!
- The vast majority of characters used are in plane 0 (BMP)
    - They only need 16 bits to represent them, giving 16 wasted bits per character
- The vast majority of characters used in the BMP are in block 1 (ASCII)
    - They only need 7 bits to represent them giving 25 wasted bits per character!!

UTF-32: is very very inefficient

“Hello 思语” ==

```
0x00000068
0x00000065
0x0000006c
0x0000006c
0x0000006f
0x00000020
0x0000601D
0x00008BED
```

8 x 4 = 32 bytes total - Look at all those leading zeros!!

UTF-8

- Goal of UTF-8 to increase efficiency
  - Waste less bits!
- Unicode has the most common characters in the first planes
  - These common characters should use less bits!
- Use variable width encoding
  - Why use 4 bytes for every character if we don’t have to?

UTF-8 Layout

#bytes	#bits	Byte 1	Byte 2	Byte 3	Byte 4
1	7	0xxxxxxx	-	-	-
2	11	110xxxxx	10xxxxxx	-	-
3	16	1110xxxx	10xxxxxx	10xxxxxx	-
4	21	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

- A single UTF-8 character can be anywhere from 1 to 4 bytes long

UTF-8 Layout

- All ASCII characters can be stored in 1 byte with zero wasted bits
- All plane 0 characters fit within 3 bytes, 8 bits more efficient than UTF-32
- Every UNICODE character can fit within 4 bytes, using exactly the same number of bits as UTF-32 in the worst case

Conversion to UTF-8 (1/2)

€ (U+20AC)

- Convert to UTF-32 (raw 32 bit representation of the code point)

  0x000020AC

  0b00000000000000000010000010101100

  - Look at all those leading zeros!
- remove leading 0s from the UTF-32 encoding

  0b10000010101100
- Split into 6 bit chunks from right to left

  `0b 10 000010 101100`

Conversion to UTF-8 (2/2)

€ (U+20AC)

- `0b 10 000010 101100`
- Match with appropriate multi-byte encoding (in this case, 3 chunks)

```
0b 1110xxxx 10xxxxxx 10xxxxxx
0b       10   000010   101100
```

- Replace the x values with the appropriate bits (0 if none)

```
0b 11100010 10000010 10101100
```

- And in hex it looks like

```
0b 1110 0010 1000 0010 1010 1100
0x    E    2    8    2    A    C
```

- We saved a byte of storage! 😃

UTF-8: More Examples

A → U+0041 → 0b01000001 → 0x41

€ → U+20AC → 0b10 000010 101100

→ 0b11100010 10000010 10101100

→ 0xE282AC

字 → U+5B57 → 0b101 101101 010111

→ 0b11100101 10101101 10010111

→ 0xE5AD97

😀 → U+1F600 → 0b 11111 011000 000000

→ 0b11110000 10011111 10011000 10000000

→ 0xF09F9880

UTF-8 - much more efficient

“Hello 思语” ==

```
0x68
0x65
0x6c
0x6c
0x6f
0x20
0xE6809D
0xE8AFAD
```

12 bytes only - and no more leading zeros!

Writing C that uses Unicode

hello_unicode.c

unicode_strings.c

utf8_strlen.c

utf8_encode.c

Summary of UTF-8

- Compact, but not minimal encoding
- ASCII is a subset of UTF-8 - complete backwards compatibility!
- no byte of multi-byte UTF-8 encoding is valid ASCII
- No byte of multi-byte UTF-8 encoding is 0
    - can still use store UTF-8 in null-terminated strings.
- 0x2F (ASCII /) and 0x00 can not appear in multi-byte characters
    - hence can use UTF-8 for Linux/Unix filenames
- C programs can treat UTF-8 similarly to ASCII
    - Beware: number of bytes in UTF-8 string != number of characters.

What we learnt Today

- Filesystems
  - C functions for reading/writing directories
  - ~
- ASCII
- Unicode
  - UTF-32 Encoding
  - UTF-8 Encoding

Next Lecture

Processes!

Reach Out

Content Related Questions:
[Forum]

Admin related Questions email:
cs1521@cse.unsw.edu.au

Student Support | I Need Help With...

My Feelings and Mental Health Managing Low Mood, Unusual Feelings & Depression	**Mental Health Connect**	student.unsw.edu.au/**counselling** Telehealth	**In Australia Call Afterhours UNSW Mental Health Support Line**	1300 787 026 5pm-9am
	Mind HUB	student.unsw.edu.au/**mind-hub** Online Self-Help Resources	**Outside Australia Afterhours 24-hour Medibank Hotline**	+61 (2) 8905 0307
Uni and Life Pressures Stress, Financial, Visas, Accommodation & More	**Student Support Indigenous Student Support**		— student.unsw.edu.au/**advisors**	
Reporting Sexual Assault/Harassment	**Equity Diversity and Inclusion (EDI)**		— edi.unsw.edu.au/**sexual-misconduct**	
Educational Adjustments To Manage my Studies and Disability / Health Condition	**Equitable Learning Service (ELS)**		— student.unsw.edu.au/**els**	
Academic and Study Skills	**Academic Language Skills**		— student.unsw.edu.au/**skills**	
Special Consideration Because Life Impacts our Studies and Exams	Special Consideration		— student.unsw.edu.au/**special-consideration**	