BASH:

- Interpreted language suitable for interaction with the OS
- **#!/bin/bash**
- **$((expression))** for math
- **$(command)** to capture program output
- **${var}** to access variables explicitly
- **$#:** number of arguments
- **bc <<< 'scale=num_digits; 100/3'** for floating point arithmetic
- **cat:** display contents of file
- **head/tail:** displays first/last 10 lines of a file
- **cut -f3 -d" "** delimits by space and gets the third field
- **sed** (sed -n 1p __ extracts 1 line)
- **tr [from] [to]:** transform input
- **sort -n -nr** (numerical)
- **if [[condition]]** \\ **then** \\ code \\ **elif** \\ **then** \\ code \\ **else** \\ code \\ **fi**
- **for** index **in** argument \\ **do** \\ *commands* \\ **done**
- **while [[condition]]** \\ **do** \\ code \\ **done**
- **1>:** send STDOUT only (same as >)
- **2>:** send STDERR only
- **&>:** send both
- **Boolean codes:** -eq, -ne, -gt, -lt, -ge, -o (or), -a (and)
- ***** matches anything
- **?** matches any single character
- **[a-z]** matches a range
- **[abc]** matches the set
- **[!a-z]** complement
- **s1 (!)= s1** for string comparison
- **/dev/null:** pipe output here to ignore it "properly"
- **&:** runs a command in the background if put after it (cannot receive STDIN)
- **fg:** brings a job back to the foreground
- **jobs:** lists all jobs started in the current terminal

C:

- Imperative, compiled, typed language
- Arguments are passed by value by default
- # calls the preprocessor
- int main(int argc, char *argv[])
- **<string.h>** functions:
  - **strlen**(const char *str) *returns size_t*
  - **strcmp**(const char *str1, const char *str2) *returns int*
  - **strcpy**(char *dest, const char *src) *returns char**
  - **memset**(void *str, int c, size_t n) *returns void**
  - **strcat**(char *dest, const char *src) *returns char**
  - **strstr**(const char *haystack, const char *needle) *returns char**
- 3 types of I/O:
  - Console
  - Stream
  - File
- int *p=(int *)malloc(sizeof(int));
- void *calloc(int numElem, int size);
- free(void *ptr);
- struct NAME {

```
struct NAME {
      FIELDS
} OPT_NAME;
```

- **typedef** requires the OPT_NAME, this will be the name of the new type
- **.c.o:**
${CC} ${CFLAGS} ${INCDIR} -c $<
tells the program how to create any needed .o file from its matching .c file (Suffix Rule)
- **int (*func)()**, syntax for function pointers

```
FILE* fp;
fp = fopen( "myfile.txt", "r" );
fseek( fp, 0L, SEEK_END );
int sz = ftell(fp);
rewind(fp);
```

```
char file_data_array[sz+1];
fread( file_data_array, 1, sz+1, fp );
printf( "File contents:\n%s\n", file_data_array );
```

- **Stack:** static memory allocation, good when amount of memory required is known before compile time
- **Heap:** dynamic memory allocation, used to allocate arbitrary amount of memory (malloc, calloc)
- **Text Space:** runnable text stored in a.out; processor loads these instructions and steps through line by line
- **Data Segment:** elements of *fixed* size that are known at compile time; stored in a.out
- **BSS (Block Started by Symbol) Segment:** space to hold global variables without initial values; filled with zeroes at runtime
- **Little Endian:** used by most systems; least significant byte comes first
- **Big Endian:** human-reading notation; most significant byte comes first (going left to right)
- Switching between Endianness does **not** change the value of data
- **LD_LIBRARY_PATH** holds all locations where libraries exist
- **export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:.**
- **CPATH:** where to search for header files
- **ldd:** prints shared object dependencies
- **nm:** list symbols from object files
- **Buffered:** most standard I/O methods are buffered
- **Unbuffered:** input is made available immediately as it comes in

Structs: padded to 4 bytes in memory

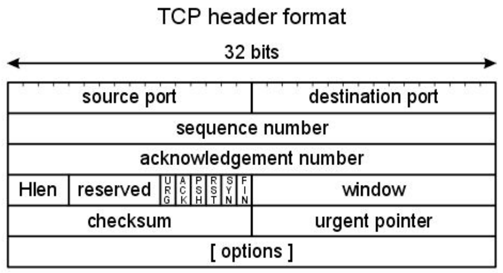
Concurrency:

```
void wait(sem S)
{
    S.count--;
    if (S.count < 0) {
        add the caller to the waiting list ;
        block();
    }
}
```

```
void post(sem S)
{
    S.count++;
    if (S.count <= 0) {
        remove a process P from the waiting list;
        resume( P);
    }
}
```

Networks:

- **Checksum:** everything in the header is added up and negated. The receiver sums everything they received and checks if adding the checksum and their sum equals zero (i.e. no data lost)
- **TCP:** Transmission Control Protocol; provides reliable, ordered and error-checked delivery of a stream of bytes between applications communicating via an IP network



- **Exponential Backoff:** if TCP detects an error it mandates that the sender send data at half the previous rate (repeat as long as there are errors)
- **TCP 3-Way Handshake:**
  - Message 1: synchronize
  - Message 2: data transfer phase
  - Message 3: terminate connection
- **C <sys/socket.h> Functions:**

- **socket();**	- **bind();**
- **listen();**	- **accept();**
- **read();**	- **write();**
- **connect();**	

- **Constant Ports:**

- 20,21: FTP	- 53: DNS
- 22: SSH/SCP	- 80: HTTP
- 25: SMTP (mail)	- 443: HTTPS

- **Socket:** connects 2 computers over a network
- **Packets:**
  - IP address of source and destination (32-bit in IPv4, 128-but in IPv6)
  - Message length
  - Time-to-live   - Header checksum
- **DNS:** name to address IP mapping (i.e. type google.com instead of 172.217.14.174)
- **DNS Dataflow:**
Client query sent -> DNS server uses tree structure to determine mapping -> asks host

server for permission to access -> receives response -> sends response to client

- HTTP: used to structure requests and responses so both sides can understand

- CGI: a protocol to allow server-side programs to be run through web interactions, producing output that drives subsequent interactions

- URL: contains server name/address, optional port number, and a path to a file on the server

- URI: the portion *after* the server address and port

- Web Browser in C:

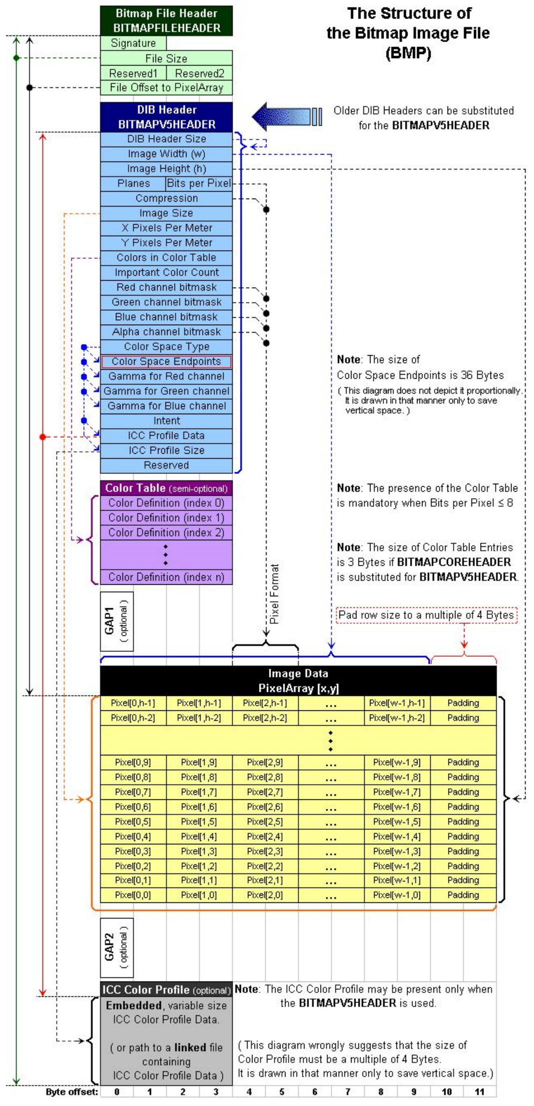
1. Initiate socket → socket()
   - Provide address and port number
2. Connect to server → connect()
3. HTTP request → “GET /HTTP1.1”
4. Write to socket → write()
5. Wait (Loop)
6. Get response from socket → read()

- HTML Form: contains name, action, method

- Action: name of script to execute
- Method: HTTP method to execute

```
<form action="cgi-bin/fist.cgi">
        <div><label><input name="message" size="20"></label></div>
        <div><input type="submit" value="Send"></div>
</form>
```

- BMP Files:



The Structure of the Bitmap Image File (BMP)
Bitmap File Header BITMAPFILEHEADER
Signature
File Size
Reserved1
Reserved2
File Offset to PixelArray
DIB Header BITMAPV5HEADER
Older DIB Headers can be substituted for the BITMAPV5HEADER
DIB Header Size
Image Width (w)
Image Height (h)
Planes
Bits per Pixel
Compression
Image Size
X Pixels Per Meter
Y Pixels Per Meter
Colors in Color Table
Important Color Count
Red channel bitmask
Green channel bitmask
Blue channel bitmask
Alpha channel bitmask
Color Space Type
Color Space Endpoints
Gamma for Red channel
Gamma for Green channel
Gamma for Blue channel
Intent
ICC Profile Data
ICC Profile Size
Reserved
Note: The size of Color Space Endpoints is 36 Bytes
(This diagram does not depict it proportionally. It is drawn in that manner only to save vertical space.)
Color Table (semi-optional)
Color Definition (index 0)
Color Definition (index 1)
Color Definition (index 2)
Color Definition (index n)
Note: The presence of the Color Table is mandatory when Bits per Pixel ≤ 8
Note: The size of Color Table Entries is 3 Bytes if BITMAPCOREHEADER is substituted for BITMAPV5HEADER.
GAP1 (optional)
Pixel Format
Pad row size to a multiple of 4 Bytes
Image Data PixelArray [x,y]
Pixel[0,h-1]
Pixel[1,h-1]
Pixel[2,h-1]
Pixel[w-1,h-1]
Padding
Pixel[0,0]
Pixel[1,0]
Pixel[2,0]
Pixel[w-1,0]
GAP2 (optional)
ICC Color Profile (optional)
Embedded, variable size ICC Color Profile Data.
(or path to a linked file containing ICC Color Profile Data)
Note: The ICC Color Profile may be present only when the BITMAPV5HEADER is used.
(This diagram wrongly suggests that the size of Color Profile must be a multiple of 4 Bytes. It is drawn in that manner only to save vertical space.)
Byte offset: 0 1 2 3 4 5 6 7 8 9 10 11

- Full Socket Code:

```c
/*
 * socket demonstrations:
 * This is the server side of an "internet domain" socket connection, for
 * communicating over the network.
 */

#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main()
{
    int fd, clientfd;
    int len;
    socklen_t size;
    struct sockaddr_in r, q;
    char buf[80];

    /* "AF_INET" says we'll use the internet */
    if ((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        return(1);
    }

    /* Specify port number. Note, we do not give our address here because
       the server's job is to listen. The whole computer has an address, so
       this is the only one we can listen on. */
    memset(&r, '\0', sizeof r);
    r.sin_family = AF_INET;
    r.sin_addr.s_addr = INADDR_ANY;
    r.sin_port = htons(1234);

    /* Bind connects the socket, tells the OS we're ready to use it. */
    if (bind(fd, (struct sockaddr *)&r, sizeof r) < 0) {
        perror("bind");
        return(1);
    }

    /* Listen blocks until a client tries to connect */
    if (listen(fd, 5)) {
      perror("listen");
      return(1);
    }

    /* Accept says, OK, let's talk! */
    size = sizeof q;
    if ((clientfd = accept(fd, (struct sockaddr *)&q, &size)) < 0) {
        perror("accept");
        return(1);
    }

    /* Read is now much like a file, we can keep grabbing data that's sent. */
    if ((len = read(clientfd, buf, sizeof buf - 1)) < 0) {
        perror("read");
        return(1);
    }
    buf[len] = '\0';
    /*
     * Here we should be converting from the network newline convention to the
     * unix newline convention, if the string can contain newlines. Ignoring for
     * now to keep it simple.
     */

    printf("The other side said: %s\n", buf);

    /* We're done listening to this client. */
    close(clientfd);

    /*
     * We didn't really have to do that since we're exiting.
     * But usually you'd be looping around and accepting more connections.
     */

    return(0);
}
```

- Libraries:

- Create/Use Static Library
  - gcc -c swap.c
  - ar rcs libswap.a swap.o
  - gcc main.c libswap.a
- Create/Use Shared Library
  - gcc -shared -fpic -o libswap.so swap.c
  - gcc main.c -lswap -L
  - Set library path:
    - Export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}
    - Environment variable

- system(): runs BASH code in a new shell

- fork(): spawns another process by copying the current one (returns negative if error, 0 in the child process, positive for the child’s process ID in the parent)

- waitpid(): waits for a specific process to finish

- Scheduling: FIFO and Round-Robin

- Scheduling Benchmarks: wait time (time between requesting and starting), latency (time to finish), throughput, fairness

Use 3: Notification

```
sem S1 = sem_init(1), S2 = sem_init(0);
       process 1                        process 2
while (1) {                       while (1) {
   // do something                   // do something
   wait(S1);         notify          wait(S2);
      printf("1");                      printf("2");
   post(S2);         notify          post(S1);
   // do something                   // do something
}                                 }
```