Sergio Garcia Tapia
Computer Systems: A Programmer's Perspective, by Bryant and O'Hallaron
Chapter 11: Network Programming
May 26, 2024

Practice Problems

Exercise 11.1. Complete the following table:

Hex address	Dotted-decimal address
0x0	_____
0xffffffff	_____
0x7f000001	_____
_____	205.188.160.121
_____	64.12.149.13
_____	205.188.146.23

Solution:

Hex address	Dotted-decimal address
0x0	0.0.0.0
0xffffffff	255.255.255.255
0x7f000001	127.0.0.1
0xcdbca079	205.188.160.121
0x400c950d	64.12.149.13
0xcdbc9317	205.188.146.23

Exercise 11.2. Write a program `hex2dd.c` that converts its hex argument to a dotted-decimal string and prints the results. For example,

```
linux> ./hex2dd 0x8002c2f2
128.2.194.242
```

Solution: My approach was to read the hexadecimal number, the use `strtol` to with a base of 16 as the third argument to parse it as a hexadecimal number in host byte order. I then proceeded to use `htonl` to convert this from host byte order to network byte order and stored it in a variable. I used the address-of operator `&` to get the pointer to this integer IP address and cast the variable to an `unsigned char *` to interpret it s a pointer to characters, where each character is a byte representation of the input number. Finally, I used `inet_ntop` to convert it to a dotted-decimal string. Notice that the book instead relied on `sscanf` instead of `strtol` to parse the input. Moreover, the book used the `struct in_addr` structure, which is the correct data structure for the IP address.

See my approach in `02-hex2dd/hex2dd.c`:

```c
#include <arpa/inet.h> /* inet_ntop(), AF_INET */
#include <stdio.h> /* fprintf(), printf(), stderr, NULL, puts() */
#include <stdlib.h> /* exit(), EXIT_FAILURE, EXIT_SUCCESS, strtol */
```

```c
#include <string.h> /* strerror() */
#include <errno.h> /* errno */

#define HEX_BASE 16
#define DOTTED_IP_LEN sizeof("255.255.255.255.255")

int
main(int argc, char *argv[])
{
    /* Validate command arguments */
    if (argc != 2) {
        fprintf(stderr, "Usage: %s hex-ipaddr\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    /* Parse hex IP address */
    char *endp;
    errno = 0;
    long hexIpAddr = strtol(argv[1], &endp, HEX_BASE);
    if (errno == ERANGE || hexIpAddr < 0 || hexIpAddr > 0xffffffff) {
        fprintf(stderr, "IP address invalid or out of range\n");
        exit(EXIT_FAILURE);
    } else if (errno != 0 || *endp != '\0') {
        fprintf(stderr, "Bad input\n");
        exit(EXIT_FAILURE);
    }

    /* Convert to network byte order */
    uint32_t networkOrderIpAddr = htonl(hexIpAddr);

    /* Interpret pointer as pointing to characters */
    unsigned char *binaryIpAddressStr = (unsigned char *) &networkOrderIpAddr;

    /* Conver to dotted IP */
    char dottedIpAddr[DOTTED_IP_LEN];
    if (inet_ntop(AF_INET, binaryIpAddressStr, dottedIpAddr, DOTTED_IP_LEN) ==
        NULL) {
        fprintf(stderr, "Invalid IP: %s\n", strerror(errno));
        exit(EXIT_FAILURE);
    }

    /* Display */
    puts(dottedIpAddr);
    exit(EXIT_SUCCESS);
}
```

Exercise 11.3. Write a program `dd2hex.c` that converts its dotted-decimal argument to a

hex number and prints the result. For example,

```
linux> ./dd2hex 128.2.194.242
0x8002c2f2
```

Solution: This time I used `inet_pton()` to parse the dotted-decimal format IP address, and I stored the result in an `unsigned char []`. I then looped through the resulting character array one byte at a time, and I used 4 iterations since an IPv4 address is made up of 4 bytes. I used the string format `%02x` to have a hex number with a width of 2 and adding 0s if necessary to pad to the full width. This produced the desired output, but the better approach is done in the book, which instead of an `unsigned char []`, they use the `struct in_inaddr` structure type, which is the correct way to interpret IP address as explained on Section 11.3.1. Then looping was not necessary because the entire number can be printed at once by using the format specifier `%x`.

See my implementation at `03-dd2hex/dd2hex.c`:

```c
#include <stdio.h> /* fprintf(), puts(), stderr */
#include <stdlib.h> /* exit(), EXIT_FAILURE */
#include <arpa/inet.h> /* inet_pton(), AF_INET */
#include <string.h> /* strerror() */
#include <errno.h> /* errno */

#define MAX_BIN_IP_LEN 65

int
main(int argc, char *argv[])
{
    /* Verify arguments were provided */
    if (argc != 2) {
        fprintf(stderr, "Usage: %s dotted-decimal-ip\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    /* Parse dotted-decimal (presentation) IP address */
    unsigned char binaryIpAddr[MAX_BIN_IP_LEN];

    switch (inet_pton(AF_INET, argv[1], binaryIpAddr)) {
        case -1:
            fprintf(stderr, "Error parsing IP address: %s\n", strerror(errno));
            exit(EXIT_FAILURE);
        case 0:
            fprintf(stderr, "Invalid IP address\n");
            exit(EXIT_FAILURE);
        case 1:
        default:
            /* Convert to hex */
            printf("0x");
```

```
            for (int i = 0; i < 4; i++)
                printf("%02x", binaryIpAddr[i]);
            printf("\n");
            break;
    }
    exit(EXIT_SUCCESS);
}
```

Exercise 11.4. The `getaddrinfo` and `getnameinfo` functions subsume the functionality of `inet_pton` and `inet_ntop`, respectively, and they provide higher-level of abstraction that is independent of any particular address format. To convince yourself how handy this is, write a version of *HOSTINFO* (Figure 11.7) that uses `inet_ntop` insted of `getnameinfo` to convert each socket address to a dotted-decimal address string.

Solution: See `04-hostinfo-inet_ntop/hostinfo_inet_ntop.c`:

```
#include <stdio.h> /* fprintf(), printf(), stderr, NULL */
#include <stdlib.h> /* exit(), EXIT_FAILURE */
#include <string.h> /* memset() */

#include <arpa/inet.h> /* inet_ntop(), AF_INET */
#include <sys/types.h> /* getaddrinfo(), freeaddrinfo(), gai_strerror(), struct
    addrinfo */
#include <sys/socket.h>
#include <netdb.h>

#define MAX_IP_ADDR_LEN sizeof("255.255.255.255")

int
main(int argc, char *argv[])
{
    struct addrinfo *p, *listp, hints;
    char buf[MAX_IP_ADDR_LEN];
    int rc;
    struct sockaddr sa;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <domain_name>", argv[0]);
        exit(EXIT_FAILURE);
    }

    /* Get a list of addrinfo records */
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = AF_INET;       /* IPv4 only */
    hints.ai_socktype = SOCK_STREAM; /* Connections only */
    if ((rc = getaddrinfo(argv[1], NULL, &hints, &listp)) != 0) {
        fprintf(stderr, "getaddrinfo error: %s\n", gai_strerror(rc));
```

```
        exit(EXIT_FAILURE);
    }

    /* Walk the list and display each IP address */
    for (p = listp; p != NULL; p = p->ai_next) {
        struct sockaddr_in *sain = (struct sockaddr_in *) p->ai_addr;
        if (inet_ntop(AF_INET, &(sain->sin_addr), buf, MAX_IP_ADDR_LEN) == NULL) {
            fprintf(stderr, "Error converting IP address: %s\n", (unsigned char
                *) (p->ai_addr));
            exit(EXIT_FAILURE);
        }
        printf("%s\n", buf);
    }

    /* Clean up */
    freeaddrinfo(listp);

    exit(EXIT_SUCCESS);
}
```

Exercise 11.5. In Section 10.11, we warned you about the dangers of using the C standard I/O functions in network applications. Yet, the CGI program in Figure 11.27 is able to use standard I/O without any problems. Why?

Solution: The example CGI program does not interleave reading from and writing to standard output; it only writes standard output. Moreover, the program does not close any of its streams explicitly; it is closed by the kernel when the child terminates. Thus there is no risk of duplicate `close` calls failing.