CSC 374/407: Computer Systems II

Lecture 8
Joseph Phillips
De Paul University

2014 August 14

Reading

- Bryant & O'Hallaron “*Computer Systems, 2nd Ed.*”
  - Chapter 10 (except 10.4): System Level I/O
  - Chapter 11: Networking Programming
- Hoover “*System Programming*”
  - Chapter 5: Input/Output

Topics

Unix Filesystem Design: A Process' Prospective

Unix Filesystem Design: A Systemwide Prospective

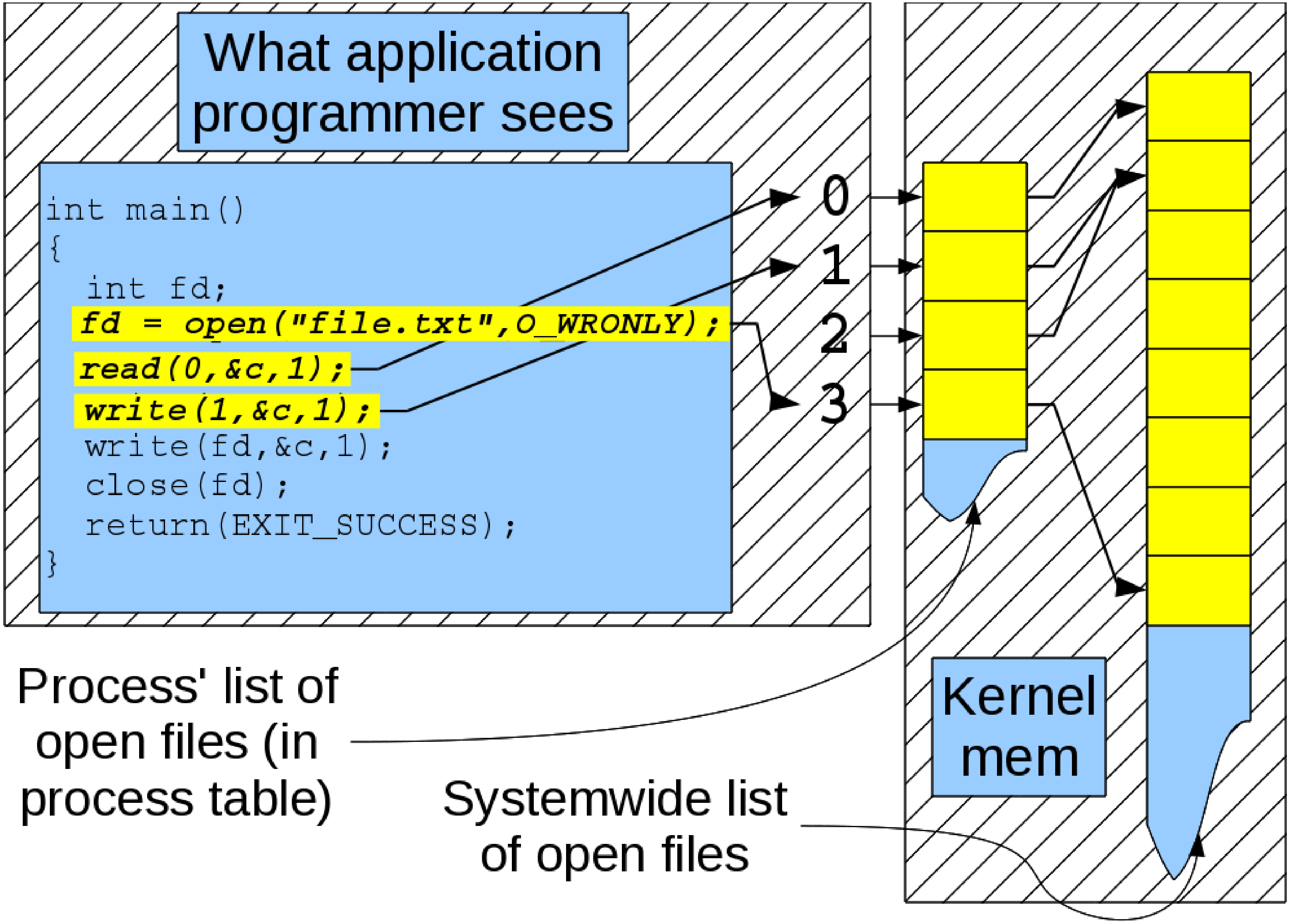
Low-level C Input-Output

Socket communication and the client/server model

Server-side socket programming

Client-side socket programming

Unix Filesystem Design: A Process' Prospective

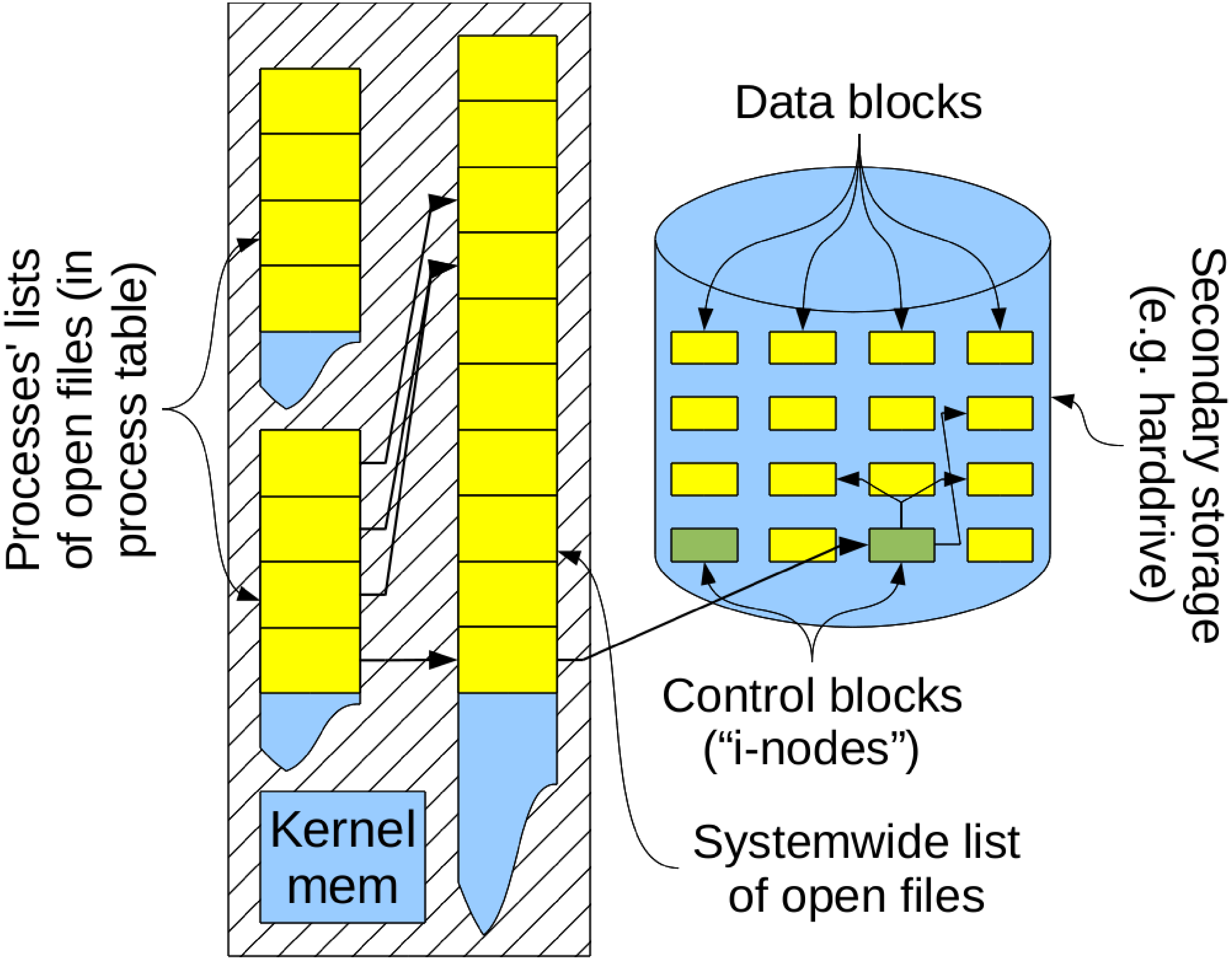


```
int main()
{
   int fd;
   fd = open("file.txt",O_WRONLY);
   read(0,&c,1);
   write(1,&c,1);
   write(fd,&c,1);
   close(fd);
   return(EXIT_SUCCESS);
}
```

Process' list of open files (in process table)

Systemwide list of open files

Unix Filesystem Design: A Systemwide Prospective

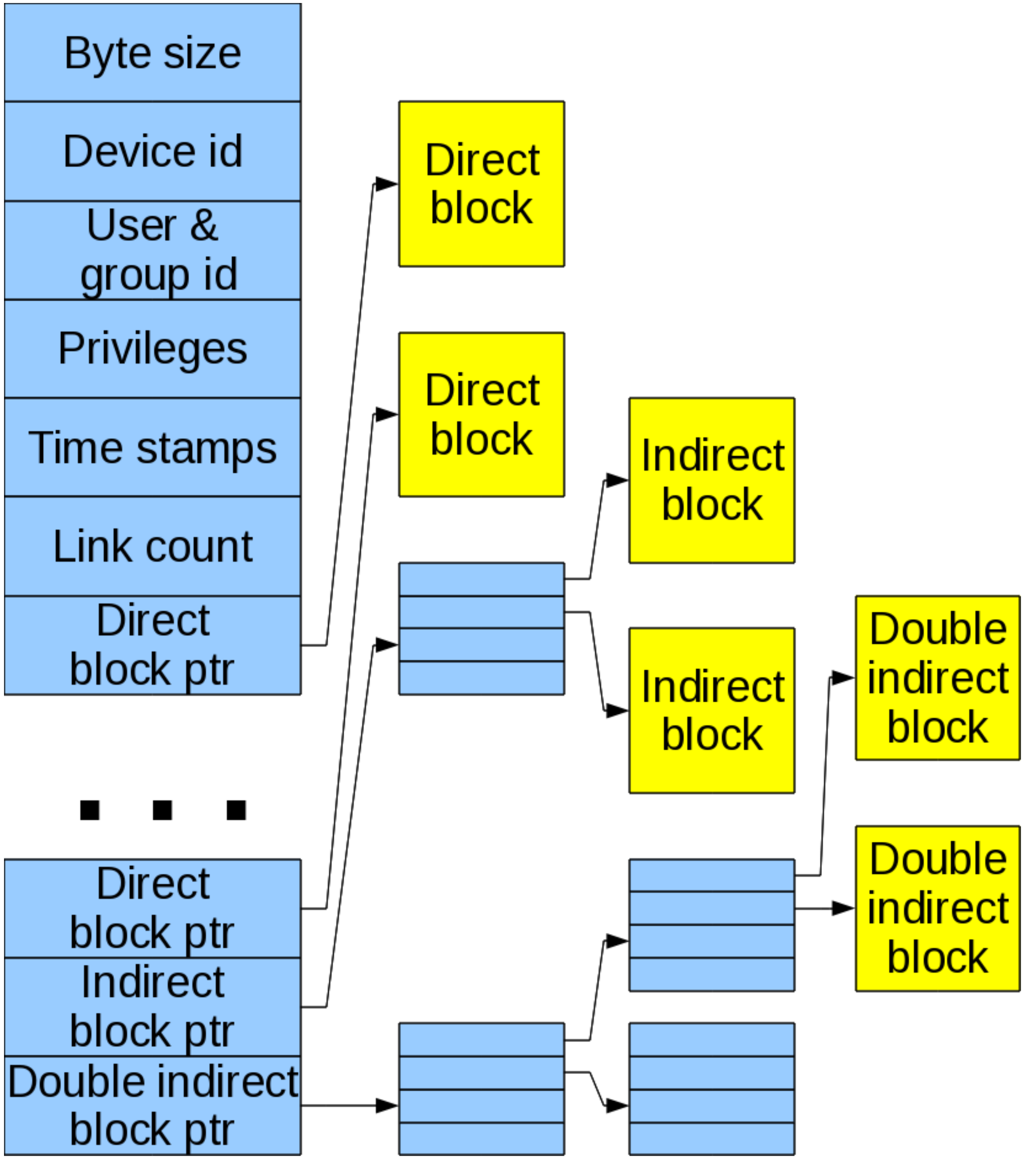


Processes' lists of open files (in process table)
Kernel mem
Data blocks
Secondary storage (e.g. harddrive)
Control blocks ("i-nodes")
Systemwide list of open files

What's an “I-Node”?

Tells a files:

- Size in bytes
- Access times (last read, last written, last its status was modified)
- User and group ID
- Device ID
- Access privileges
- Link count (num different names/directories)
- Pointers



Low level C Input-Output

File descriptors are indices into process' file table

- 0: Standard input (***stdin***)
- 1: Standard output (***stdout***)
- 2: Standard error (***stderr***)

Useful commands include:

```
int open(const char* path, int how,
  int permission)
int close(int fd)
int read(int fd, char* bufferPtr,
  size_t bufferSize)
int write(int fd, char* bufferPtr,
  size_t numBytes)
int dup(int fd);
int pipe(int** );
```

open()

*`int open(`**`const char* path`**`, int how, int permission)`*

- Returns file descriptor (index into process' file array)
- File path given by `path`.

open()

```
int open(const char* path, int how,
  int permission)
```

- Integer how is bitwise or-ing of one of:
  - `O_RDONLY`: Open for reading only.
  - `O_WRONLY`: Open for writing only.
  - `O_RDWR`: Open for reading and writing.
- And perhaps one or more of:
  - `O_CREAT`: Create file if doesn't already exist
  - `O_TRUNC`: If exist truncate its length to 0 (even if not open for writing)
  - `O_EXCL`: If `O_CREAT` is also set fail if file exists.
  - `O_APPEND`: Write to end of file.

open(), cont'd

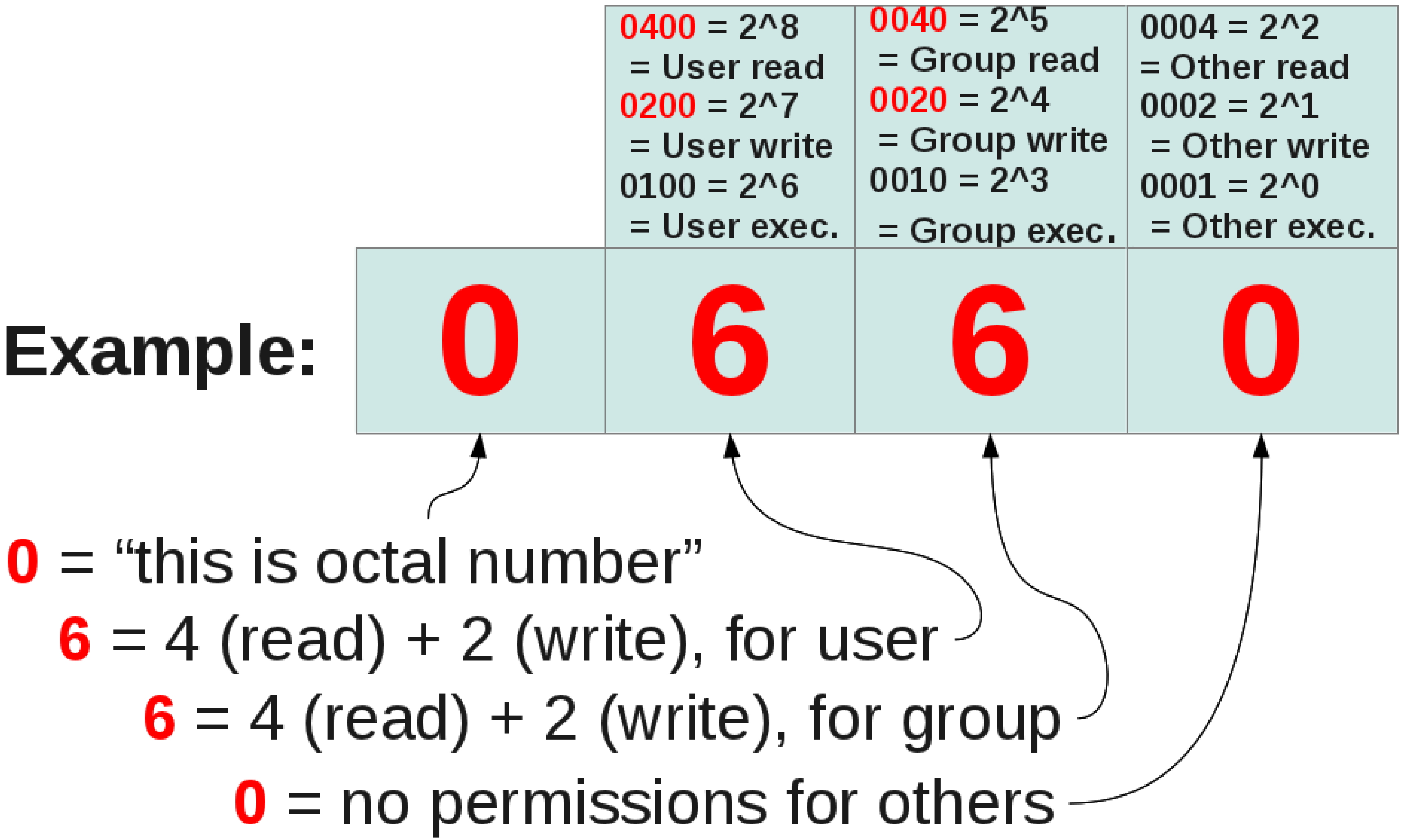
```
int open(const char* path, int flags,
  int permission)
```

- Commonly specified in *octal*: “0” + $d_2 {*} 8^2 + d_1 {*} 8^1 + d_0 {*} 8^0$
  - ***0x***10: leading “0x” mean “hexadecimal”
  - ***0***20: leading “0” mean “octal”

000 (0)	001 (1)	002 (2)	003 (3)	004 (4)	005 (5)	006 (6)	007 (7)
010 (8)	011 (9)	012 (10)	013 (11)	014 (12)	015 (13)	016 (14)	017 (15)
??? (?)	??? (?)	??? (?)	??? (?)	??? (?)	??? (?)	??? (?)	??? (?)

- Digits: 04 = read, 02 = write, 01 = execute permission
- Place: 64s pos. = user, 8s pos. = group, 1s pos. other
- Permissions are the bitwise or-ing (|) of permissions for read/write/execute for user/group/other

open(), cont'd



	0400 = 2^8 = User read 0200 = 2^7 = User write 0100 = 2^6 = User exec.	0040 = 2^5 = Group read 0020 = 2^4 = Group write 0010 = 2^3 = Group exec.	0004 = 2^2 = Other read 0002 = 2^1 = Other write 0001 = 2^0 = Other exec.
Example: 0	6	6	0

0 = “this is octal number”

6 = 4 (read) + 2 (write), for user

6 = 4 (read) + 2 (write), for group

0 = no permissions for others

YOUR TURN

(1) What is the **octal code** for:
user: read, write;
group: read-only;
other: read-only

(2) What does octal code **0750** allow the user, group and everyone else to do?

`close()`

`int close (int fd)`

- Closes `fd`.
- Returns `0` on success or `-1` otherwise.
- Does not flush file.

Your turn!

You are going to open a process' *first file* after standard input (0), standard output (1) and standard error (2). The file descriptor is an index in a table. What is its integer value?

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>

int main ()
{
  int fd = open("bubu.txt",O_WRONLY|O_CREAT|O_APPEND,0660);
  printf("fd = %d\n",fd);
  close(fd);
  return(EXIT_SUCCESS);
}
```

write()

```
int write(int fd, char* bufferPtr,
  size_t numBytes)
```

- Writes `numBytes` pointed to by `bufferSize` to `fd`.
- Returns number of bytes written, or `-1` on error.

write() example

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>

int main ()
{
  int fd = open("bubu.txt",O_WRONLY|O_CREAT|O_APPEND,0660);
  write(fd,"Bubu!\n",6);
  close(fd);
  system("ls -l ./bubu.txt");
  return(EXIT_SUCCESS);
}
```

`read()`

`int read(int fd, char* bufferPtr, size_t bufferSize)`

- Reads up to `bufferSize` bytes from `fd` and puts them into `bufferPtr`.
- Returns number of bytes read from file, either
  - `0` (“*No more left!*”),
  - `bufferSize` (“*Here's a whole buffer full!*”),
  - somewhere inbetween (“*Here's all that's left*”), or,
  - `-1` (“*Error!*”)

read() example

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>

#define       BUFFER_SIZE     256

int main ()
{
  char buffer[BUFFER_SIZE];
  int  fd = open("bubu.txt",O_RDONLY,0660);

  read(fd,buffer,BUFFER_SIZE);
  printf("%s",buffer);
  close(fd);
  return(EXIT_SUCCESS);
}
```

Your turn!

Write your own *simple* version of the Unix `littleCopy` file copying command. I'll get you started:

```
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>

#define       BUFFER_SIZE     256

/* Continued on next slide */
```

Your turn!

```
/* From previous slide */

int main (int argc, const char* argv[])
{
  const char* fromFileCPtr;
  const char* toFileCPtr;

  if  (argc < 3)
  {
    fprintf(stderr,
            "Usage: littleCopy <fromFile> <toFile>\n"
           );
    return(EXIT_FAILURE);
  }

  fromFileCPtr = argv[1];
  toFileCPtr   = argv[2];

  /* YOUR CODE HERE */
  return(EXIT_SUCCESS);
}
```

Your turn, again!

```
/* Write a program that counts the number of occurrences
   of a character given on the command line. */
int main (int argc, const char* argv[])
{
  const char  charToCount;
  const char* fileCPtr;

  if  (argc < 3)
  {
    fprintf(stderr,
            "Usage: charCount <char> <file>\n"
           );
    return(EXIT_FAILURE);
  }

  charToCount = *argv[1];
  fileCPtr    = argv[2];

  /* YOUR CODE HERE */
  return(EXIT_SUCCESS);
}
```

And your turn, yet again!

Revise the previous program to
count the number of lines in a file.

What happens if mother and child write to same file?

```
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>

#define FILENAME  "bubu.txt"

int     main    ()
{
  const char*   wordsPtr;
  int  i;
  int  numBytes;
  int  pid;
  int  fd =
  open(FILENAME,
       O_WRONLY|O_CREAT|O_TRUNC,
       0660);
```

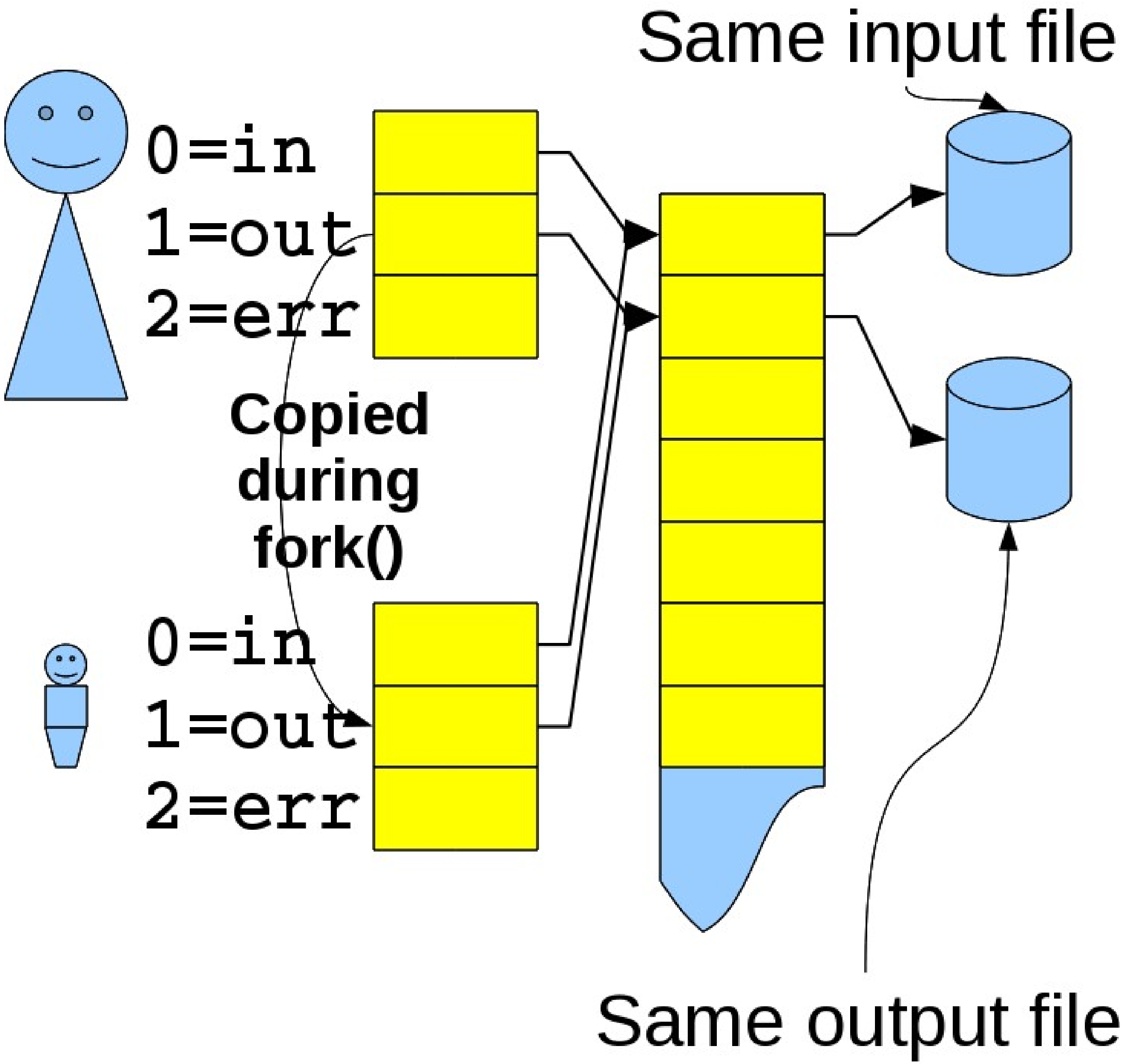
```
if  (fd < 0)
{
  fprintf(stderr,
  "Sorry, I can't make "
  "the output file %s\n",
  FILENAME);
  return(EXIT_FAILURE);
}

pid = fork();

if  (pid < 0)
{
  fprintf(stderr,
  "Too many processes ace!\n"
  );
  return(EXIT_FAILURE);
}
```

What happens if mother and child write to same file?

```
else
if  (pid == 0)
  wordsPtr =
    "Baby says \"Gaga Gugu!\"\n";
else
  wordsPtr =
    "Mama says \"Poor baby!\"\n";

numBytes = strlen(wordsPtr);
```

```
  for  (i = 0;  i < 4;  i++)
  {
    sleep(1);
    write(fd,wordsPtr,
          numBytes);
    printf(wordsPtr);
  }

  if  (pid > 0)
  {
    sleep(2);
    close(fd);
  }

  return(EXIT_SUCCESS);
}
```

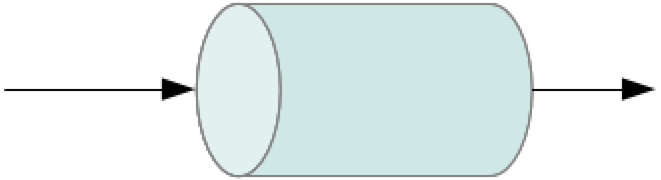
What's going on?



Hey! Maybe we can use this for interprocess communication!

```
#include <unistd.h>
. . .
const int PIPE_READ = 0;
const int PIPE_WRITE= 1;
int       myPipe[2];

if  (pipe(myPipe) == 0)
{
  char myArray[6];
  write(myPipe[PIPE_WRITE],"Hello!",6);
  read (myPipe[PIPE_READ ],myArray, 6);
}
```

"Hello!"
into
myPipe[1]



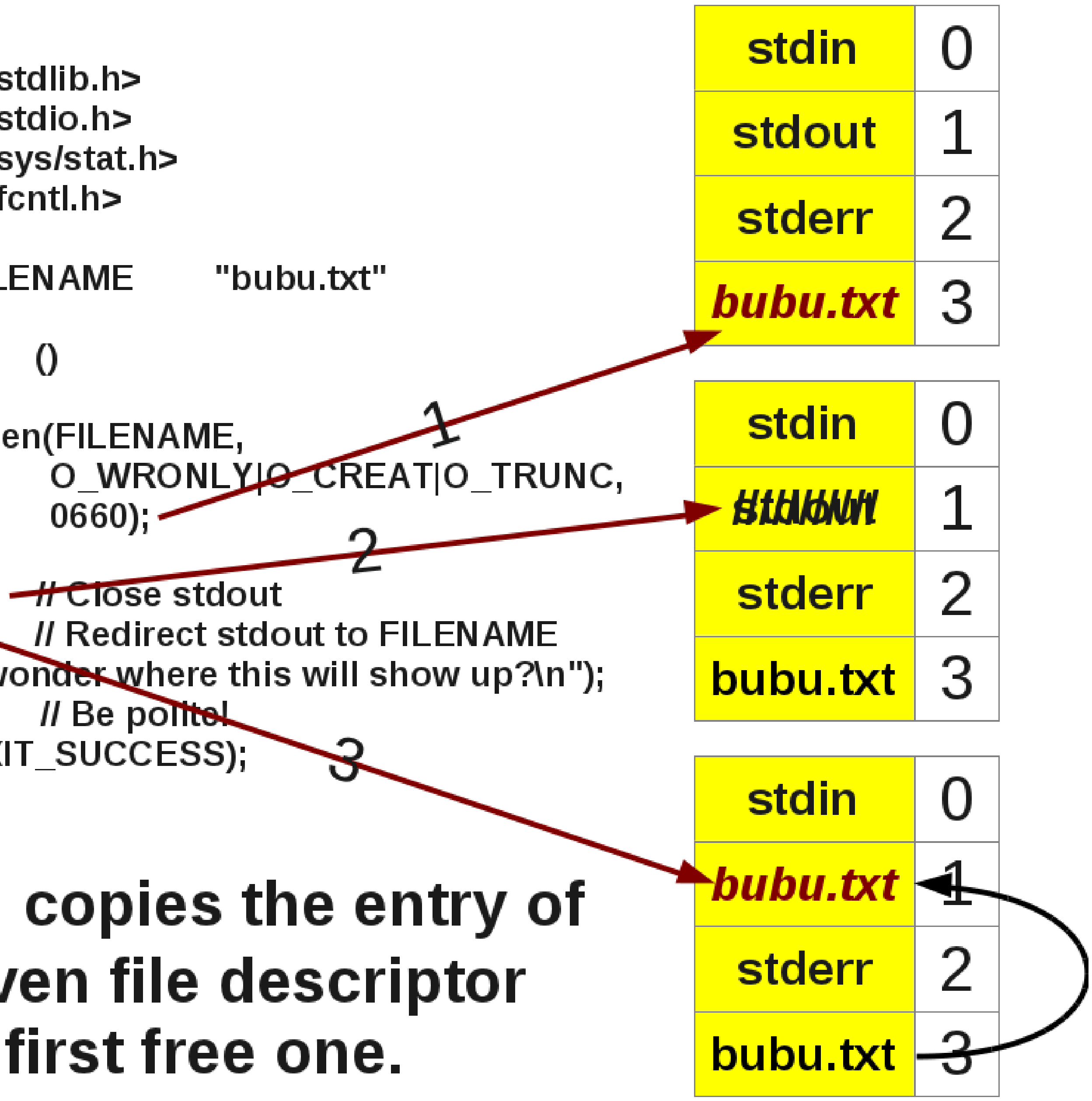
"Hello!"
out from
myPipe[0]

myPipe: An OS-owned buffer

dup()

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>

#define FILENAME        "bubu.txt"

int     main    ()
{
  in  fd= open(FILENAME,
                        O_WRONLY|O_CREAT|O_TRUNC,
                        0660);

  close(1);    // Close stdout
  dup(fd);     // Redirect stdout to FILENAME
  printf("I wonder where this will show up?\n");
  close(fd);    // Be polite!
  return(EXIT_SUCCESS);
}
```

`dup()` copies the entry of the given file descriptor to the first free one.



stdin	0
stdout	1
stderr	2
bubu.txt	3

stdin	0
~~stdout~~	1
stderr	2
bubu.txt	3

stdin	0
bubu.txt	1
stderr	2
bubu.txt	3

IPC with pipes

```
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>

int     main    ()
{
  int   parentToChild[2];
  int   childToParent[2];

  if  ((pipe(parentToChild)  < 0)
       ||(pipe(childToParent)< 0))
  {
    fprintf(stderr,
            "Can't make pipes\n");
    return(EXIT_FAILURE);
  }

  int   pid     = fork();
```

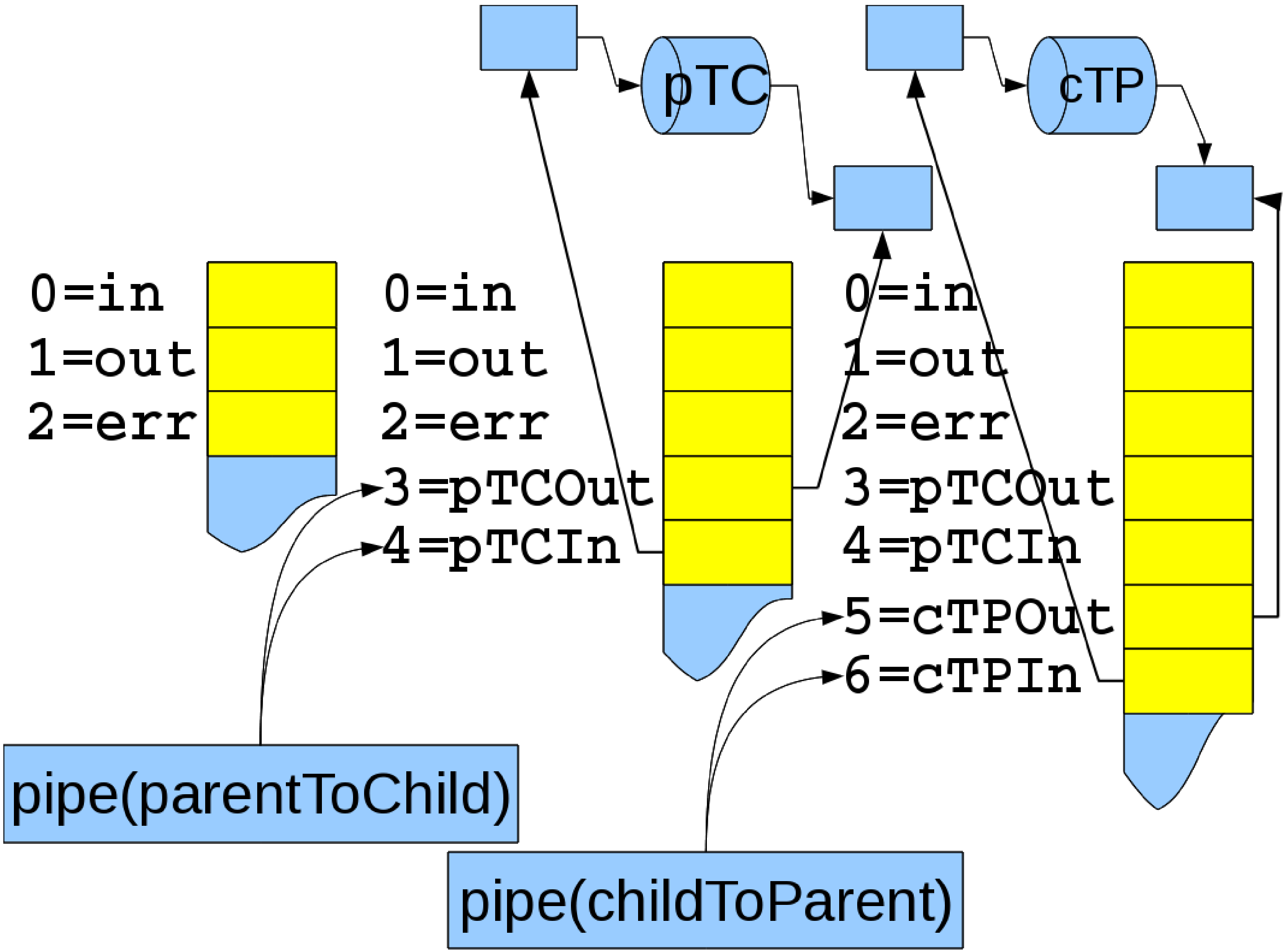
```
if  (pid < 0)
{
  fprintf(stderr,"Too many
processes Ace!\n");
  return(EXIT_FAILURE);
}
else
if  (pid == 0)
{

  //  Baby's case
  close(0);   // Close "stdin"
  dup(parentToChild[0]);
  close(1);   // Close "stdout"
  dup(childToParent[1]);

  // . . . continued
```

IPC with pipes, cont'd

```
  // Baby's case, continued
  while (1)
  {
    char buffer[10];
    int i,numRead;
    numRead =
          read(0,buffer,10);

    for (i=0; i<numRead; i++)
      buffer[i] =
        toupper(buffer[i]);

    write(1,buffer,numRead);
  }
}
else
```

```
  {
    // Mama's case
    while (1)
    {
      char buffer[10];
      fgets(buffer,10,stdin);
      write
        (parentToChild[1],
         buffer,
         10);
      read
        (childToParent[0],
         buffer,
         10);
      printf(buffer);
    }
  }
  return(EXIT_SUCCESS);
}
```

dup() and pipe(), 1



pTC

cTP

0=in
1=out
2=err

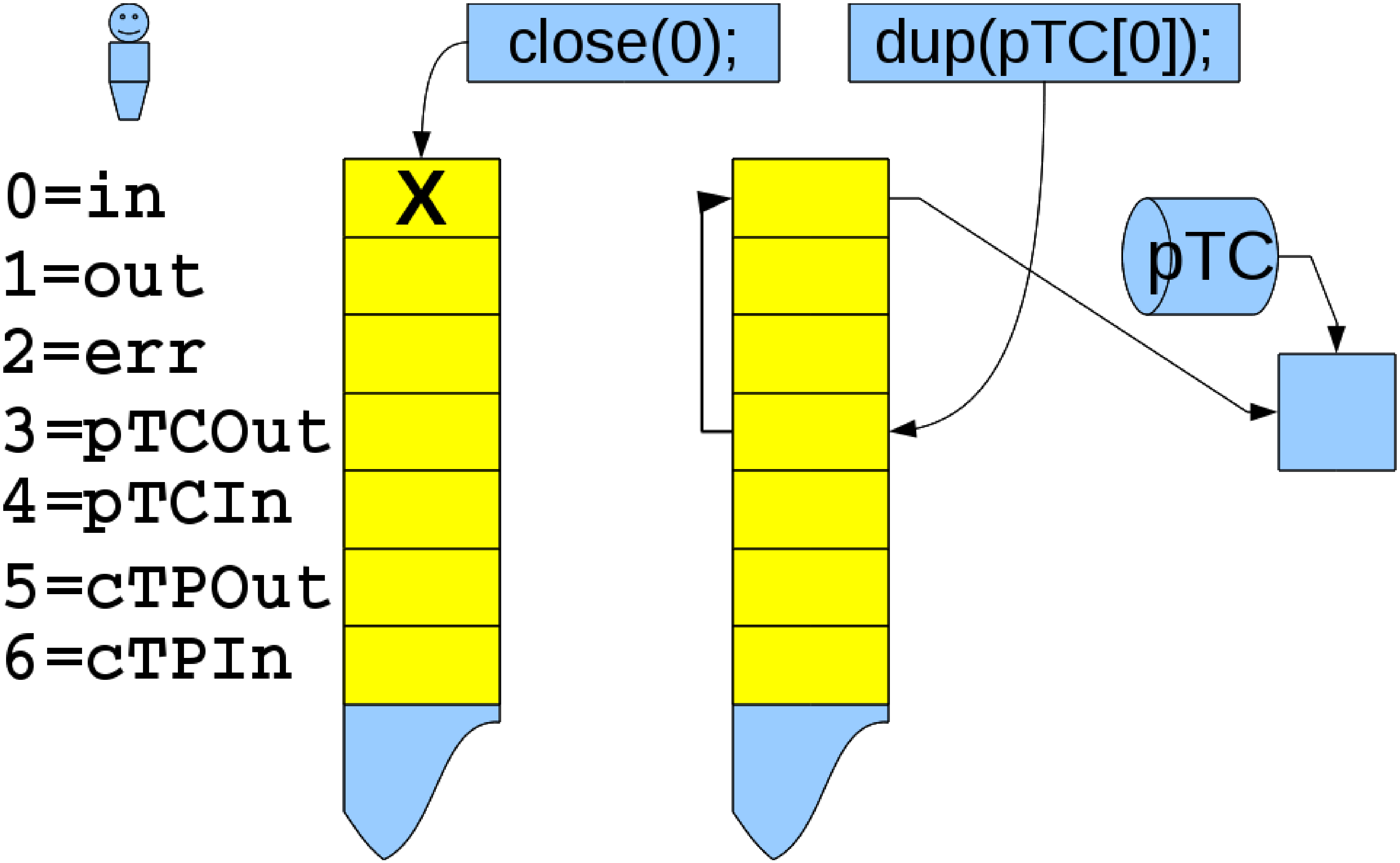
0=in
1=out
2=err
3=pTCOut
4=pTCIn

0=in
1=out
2=err
3=pTCOut
4=pTCIn
5=cTPOut
6=cTPIn

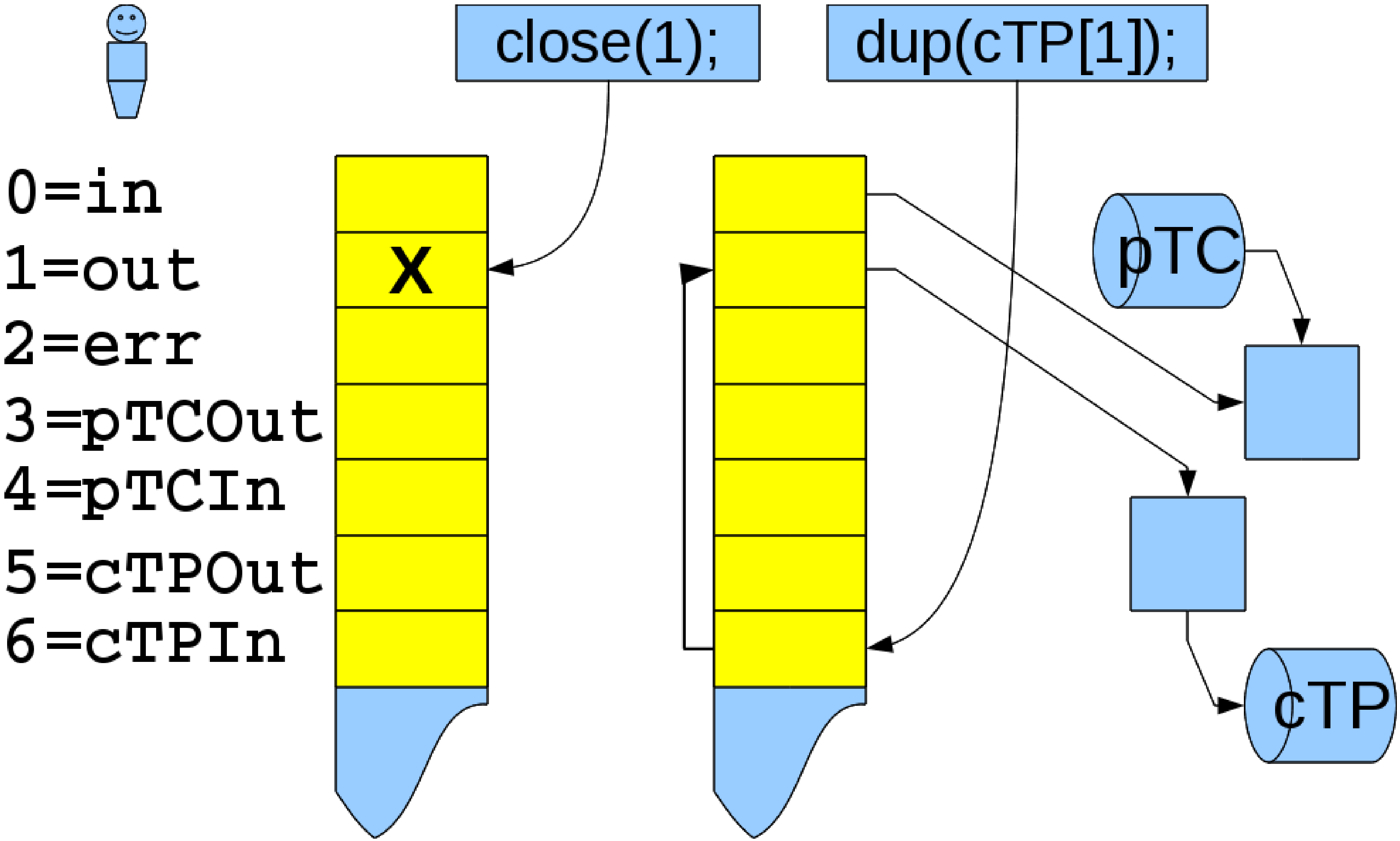
pipe(parentToChild)

pipe(childToParent)

dup() and pipe(), 2



dup() and pipe(), 3

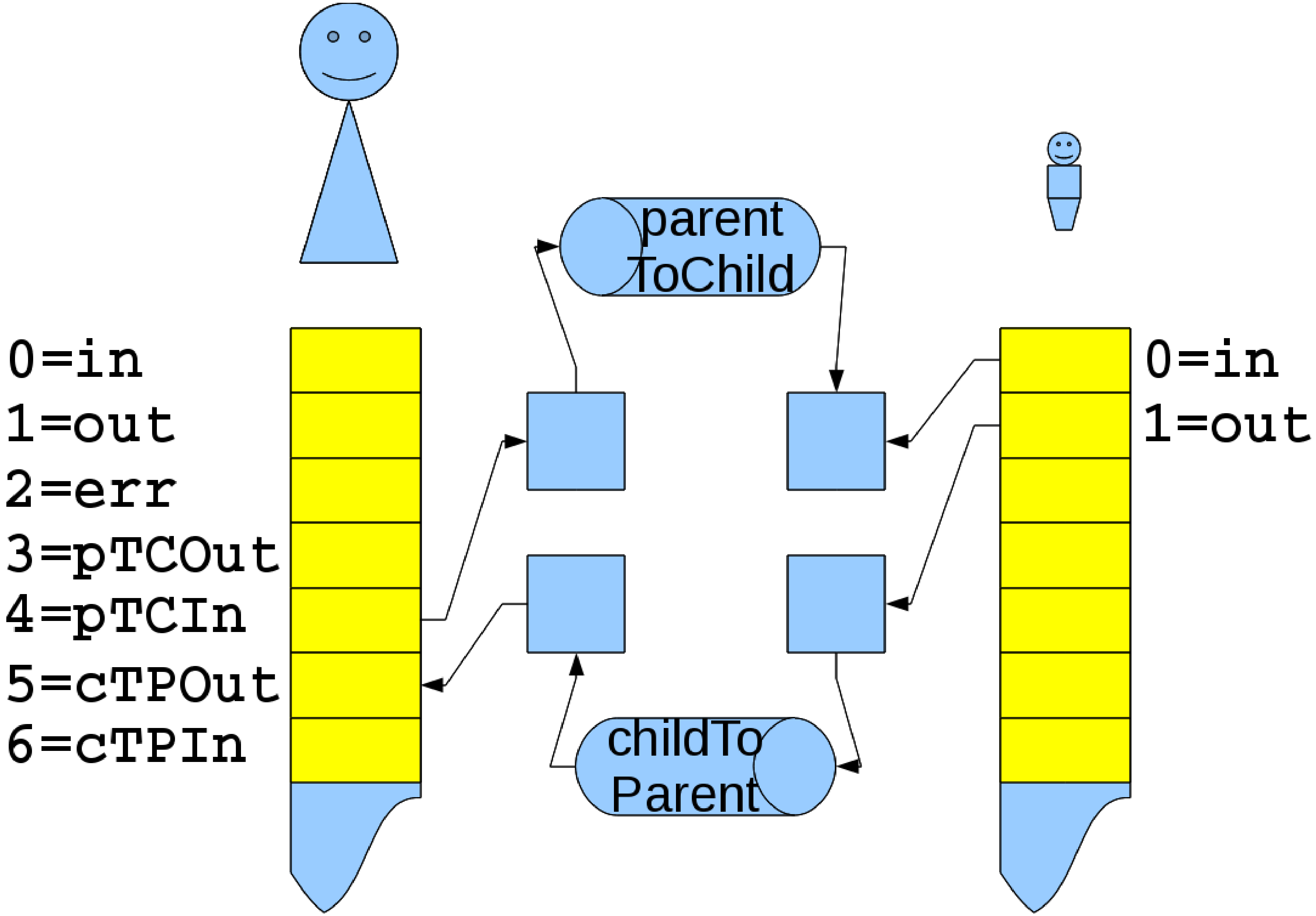


0=in
1=out
2=err
3=pTCOut
4=pTCIn
5=cTPOut
6=cTPIn

close(1);

dup(cTP[1]);

dup() and pipe(), 4



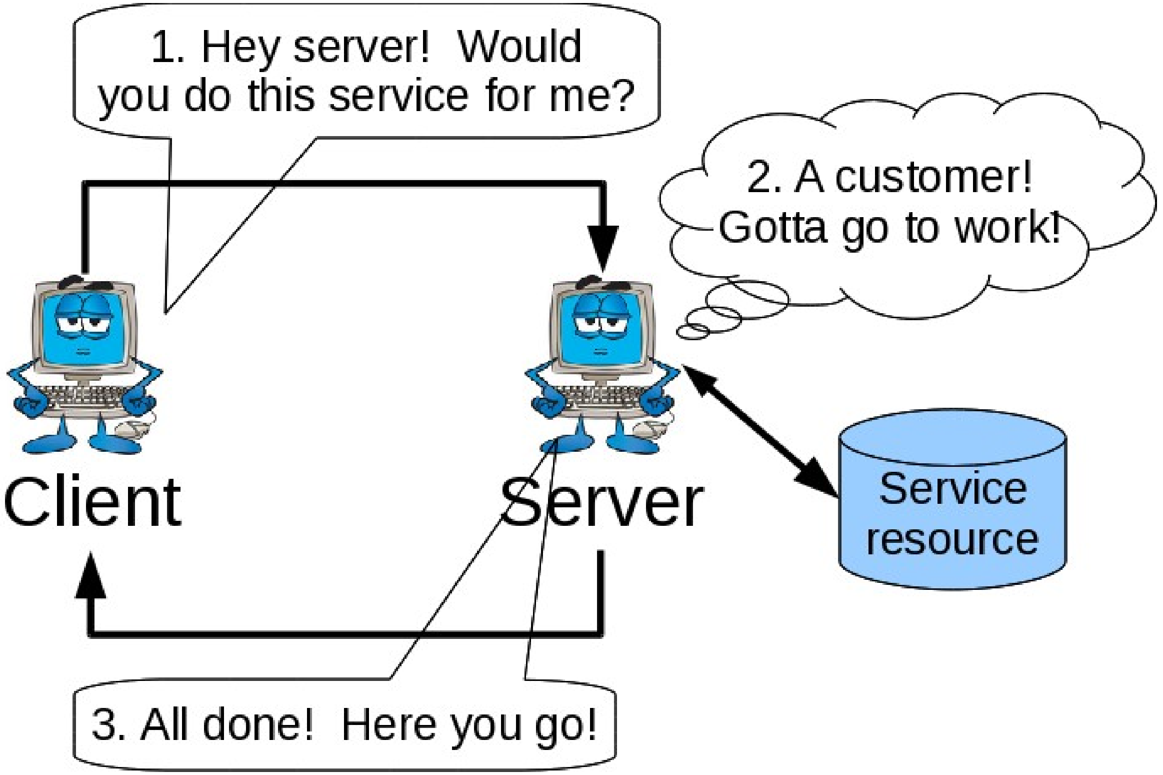
Sockets

- Hey! Moving bytes to/from a file descriptor is such a ***grand idea*** that we can use it to move bytes to/from ***another process***
- Further, the process could be ***here*** (on the same machine)
- . . . .
  . . . . or way ***over here*** (another machine across the Internet)
- We call this ***sockets!***

The client-server computing model

1. Client asks server for a service
2. Server does service
3. Server returns result to client

Examples: `ssh`, `sftp`, `http`, *etc*.



1. Hey server! Would you do this service for me?
2. A customer! Gotta go to work!
Client
Server
Service resource
3. All done! Here you go!

Processes talking to each other on different computers

Identify service by ***IP address*** and ***port***

- IP address: ***Which computer?***
  - Humans like strings: "*www.depaul.edu*"
  - Computers like numbers: *75.102.246.202*
- DNS: *D*omain *N*ame *S*ervice
  - Given name get number (or vice versa)
- Computers refer to themselves by the "loopback address"
  - *127.0.0.1* (integers)
  - *localhost* or *localhost.localdomain* (string)

Ports:

Ports: ***Which service on the given computer?***

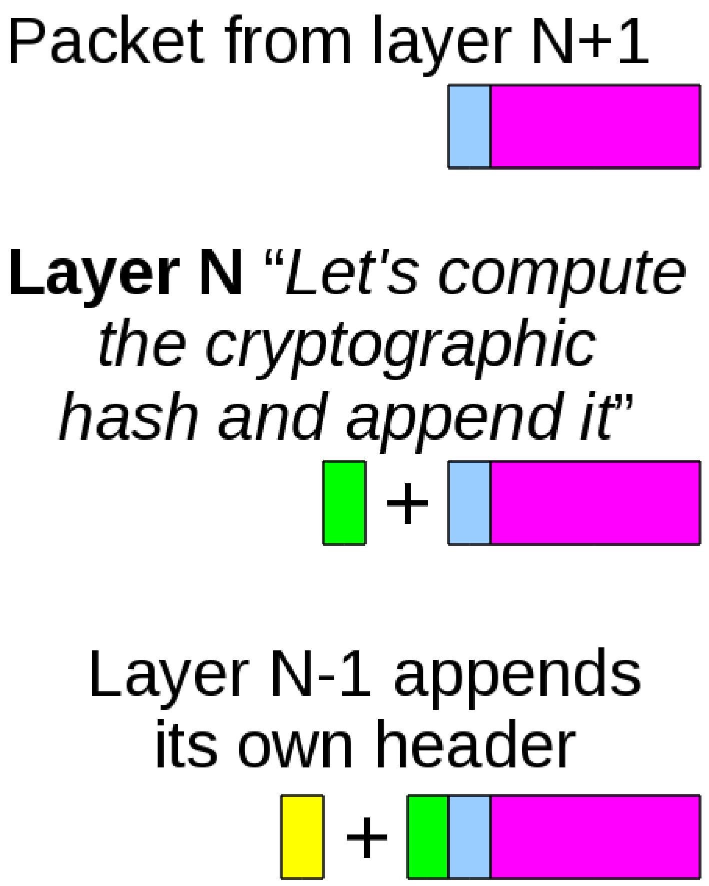
Can range from 0 . . . 65535?

Common ones:

- 20 (ftp data), 21 (ftp control)
- 22 (ssh), 23 (telnet <-- DO NOT USE TO LOG IN!)
- 37 (time)
- 80 (www/http)

Packets (or datagrams)

- Any communication is split up into manageable chunks (“packets”) that are sent individually.
- These chunks get routing, checksum, cryptographic, *etc.* info added to them

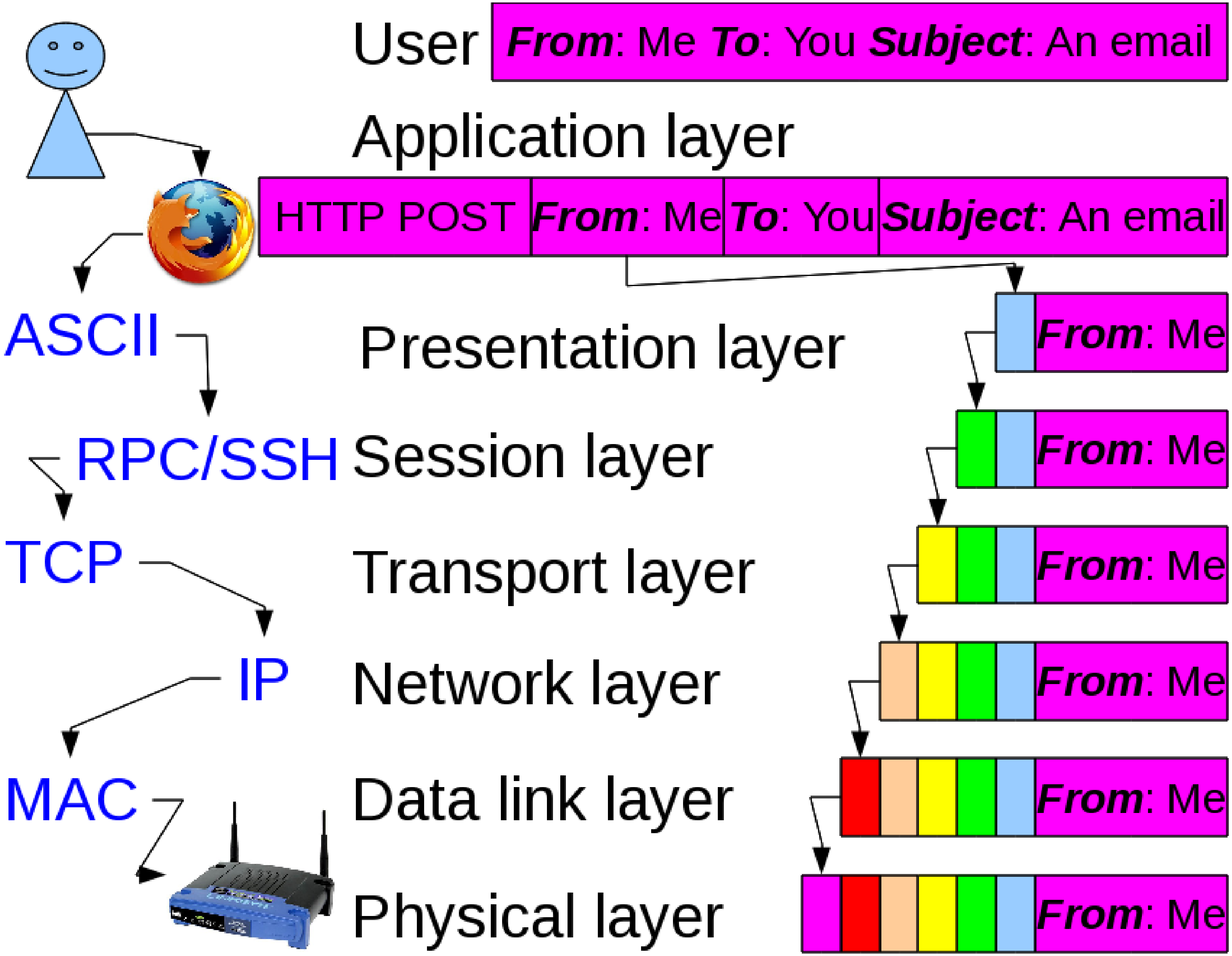


Packet from layer N+1

Layer N “*Let's compute the cryptographic hash and append it*”

Layer N-1 appends its own header

So I know with whom I wanna talk, how do I communicate? (1)



User

From: Me ***To***: You ***Subject***: An email

Application layer

HTTP POST ***From***: Me ***To***: You ***Subject***: An email

ASCII

Presentation layer

From: Me

RPC/SSH

Session layer

From: Me

TCP

Transport layer

From: Me

IP

Network layer

From: Me

MAC

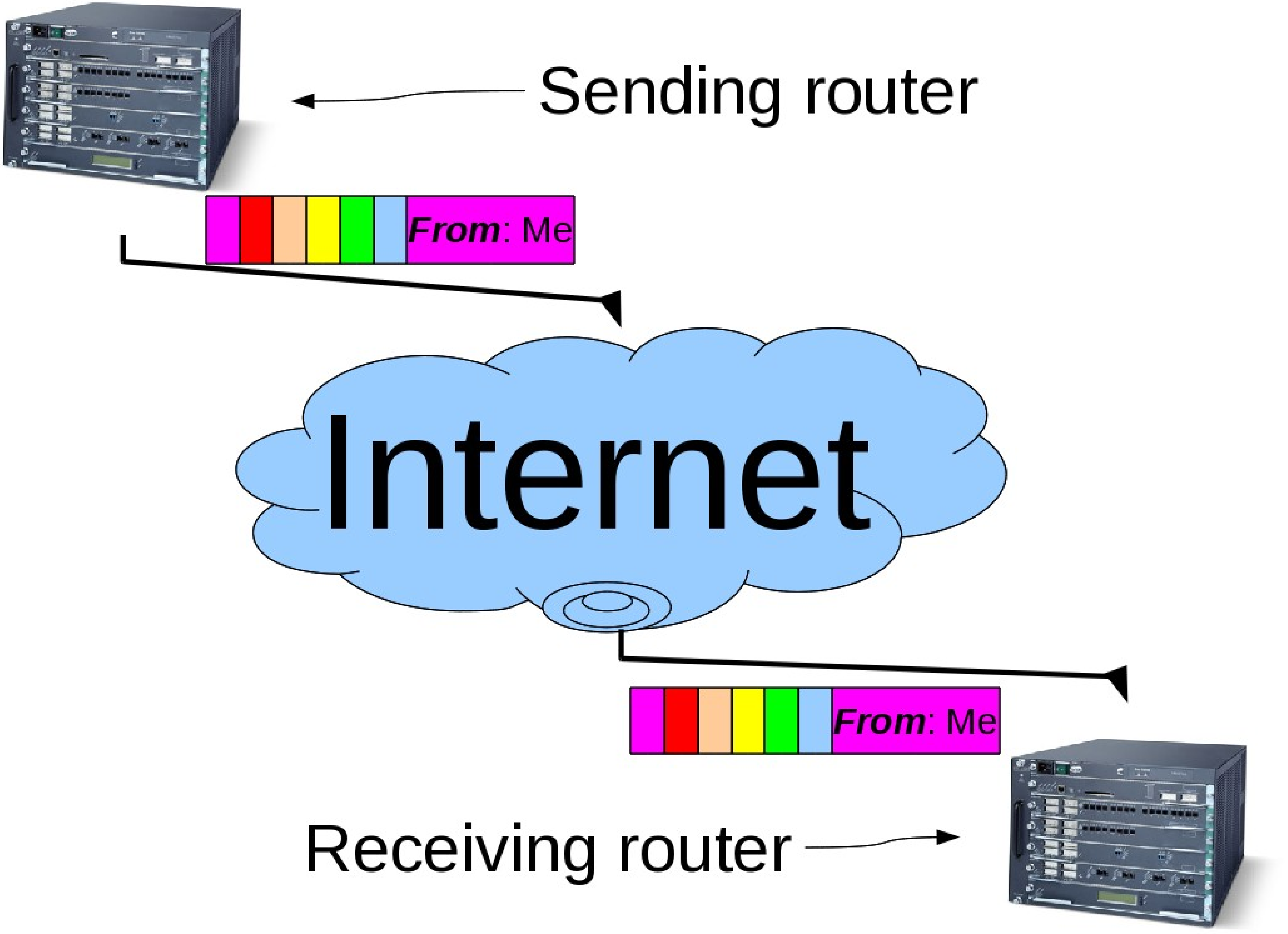
Data link layer

From: Me

Physical layer

From: Me

So I know with whom I wanna talk, how do I communicate? (2)



Sending router

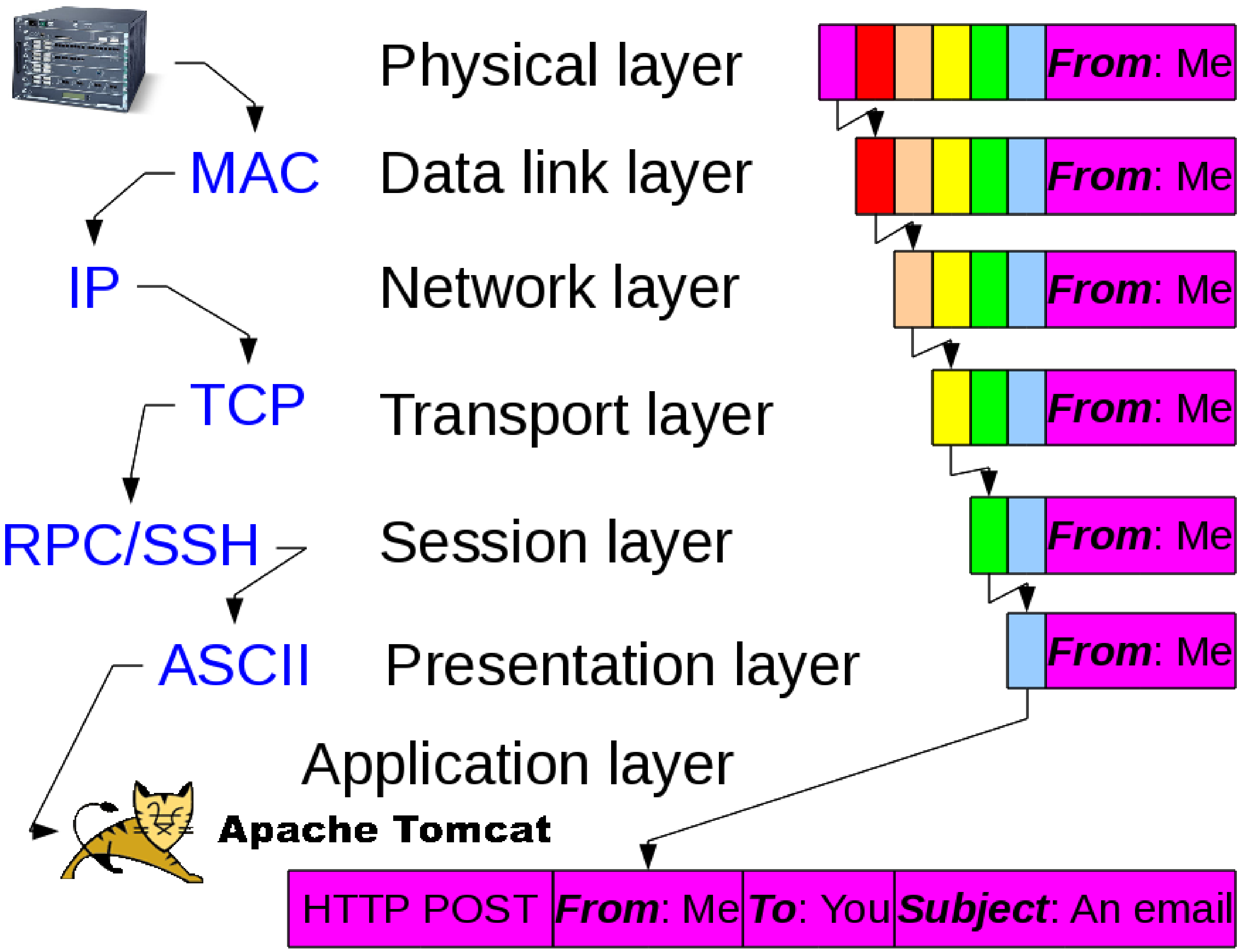
From: Me

Internet

From: Me

Receiving router

So I know with whom I wanna talk, how do I communicate? (3)



Physical layer — ***From***: Me

MAC — Data link layer — ***From***: Me

IP — Network layer — ***From***: Me

TCP — Transport layer — ***From***: Me

RPC/SSH — Session layer — ***From***: Me

ASCII — Presentation layer — ***From***: Me

Application layer

HTTP POST | ***From***: Me | ***To***: You | ***Subject***: An email

Tell me more about packets being sent over the 'net!

Two common networking protocols:

- UDP: *User Datagram Protocol*
  - Fast! Unreliable!
  - "*Just send the packets! Missed the last one? Don't worry, here's another!*"
  - **Question: When would you use this?**
- TCP: *Transmission Control Protocol*
  - Slower! Transmission is verified!
  - "*Just a received a packet. Slow down! Let's make sure it hasn't been corrupted and that I have them all.*"
  - **Question: When would you use this?**

Sockets!

Question: Cool deal! How do I program it?

Answer: With sockets, silly!

Socket communication

- Done with `read()` and `write()` (just like for files)
- Both have their own <address,port> pair
- Socket provides non-transient 2-way communication link

Server Side:

```
#include <sys/socket.h>//For socket()
#include <netinet/in.h>//For sockaddr_in and htons()
#include <netdb.h>     //For getaddrinfo()
#include <errno.h>     //For errno var
#include <sys/stat.h>  //For open(), read(),write()
#include <fcntl.h>     // and close()
```

- socket(): Ask OS for a socket
- bind(): Bind socket and port together
- listen(): Tell how many clients may queue
- accept(): Wait until a client connects
- write(): Write to client/server
- read(): Read from client/server
- close(): Close socket with client/server.

Client Side:

```
#include <sys/socket.h>//For socket()
#include <netinet/in.h>//For sockaddr_in and htons()
#include <netdb.h>     //For getaddrinfo()
#include <errno.h>     //For errno var
#include <sys/stat.h>  //For open(), read(),write()
#include <fcntl.h>     // and close()
```

- `getaddrinfo()`: Find server's IP address
- `socket()`: Ask OS for a socket
- `connect()`: Attempt to connect to server
- `write()`: Write to server
- `read()`: Read from server
- `close()`: Close socket with server.

socket()

```
// Create a socket
int socketDescriptor =
          socket(AF_INET,      // AF_INET domain
                 SOCK_STREAM, // Reliable TCP
                 0);
```

- Returns
  - A file descriptor that the server uses to see if a client has connected, or,
  - `-1` on error
- There's also `SOCK_DGRAM` for UDP
- Last parameter type if used for `SOCK_RAW`

bind()

```
// Bind socket to port
// We'll fill in this datastruct
struct sockaddr_in socketInfo;

// Fill socketInfo with 0's
memset(&socketInfo,'\0',sizeof(socketInfo));
// Use std TCP/IP
socketInfo.sin_family = AF_INET;
// Tell port in network endian with htons()
socketInfo.sin_port = htons(portNumber);
  // (1) Allow connections from myself only:
  struct in_addr addr;
  if (inet_aton("127.0.0.1",&addr)==0)  exit(EXIT_FAILURE);
  socketInfo.sin_addr.s_addr = addr.s_addr;
  // or (2) Allow machine to connect to this service
  socketInfo.sin_addr.s_addr = INADDR_ANY;
//  Try to bind socket with port and other specifications
int status = bind(socketDescriptor, // from socket()
                  (struct sockaddr*)&socketInfo,
                  sizeof(socketInfo));
status == -1 on error
```

What are those structs?

```
typedef uint32_t in_addr_t;

struct in_addr
{
  in_addr_t s_addr;
};

struct sockaddr_in
{
  sa_family_t    sin_family; // addr family: AF_INET
  in_port_t      sin_port;   // port (in network
                             //  byte order)
  struct in_addr sin_addr;   // internet addr
};
```

listen()

```
// Tell OS how many clients may queue
  up for this server
int status =
        listen(socketDescriptor,
               maxNumPendingClients);
```

- (Almost) ready to listen to port!
- 5 is a good default for `maxNumPendingClients`.
- If `status==-1` then error

accept()

```
// Accept connection to client
int clientDescriptor =
     accept(socketDescriptor,NULL,NULL);
```

- Wait (by default) for someone to actual connect
- Returns
  - a file descriptor for talking with one particular client, or
  - `-1` for error
- `connectionDescriptor` for talking with that one client (there may be others for other clients)
- `socketDescriptor` is for listening to socket.

Your turn!

Question: **Hey!** How is the server supposed to do two (or more!) things at once?

How do we get the server to both:

1. wait for another client to connect by listening to `socketDescriptor`, and
2. handle the current client(s) request by talking on `clientDescriptor`?

Do you speak ***BIG*** or ***little*** Endian?

Now that we're talking . . . we'd better use same endian!

```
// Host to network long (ie. 32-bit)
uint32_t htonl(uint32_t hostlong);

// Host to network short (ie. 16-bit)
uint16_t htons(uint16_t hostshort);

// Network to host long (ie. 32-bit)
uint32_t ntohl(uint32_t netlong);

// Network to host short (ie. 16-bit)
uint16_t ntohs(uint16_t netshort);
```

Your turn again!

- Write a server program that

1. waits for a client to connect
2. for any connected client it `read()`s characters and `write()`s the `toupper()` of them.

Client-side time!

getaddrinfo()

// Get info on server (the “hostName”)

```
int getaddrinfo
(const char*   hostName, // e.g. "www.depaul.edu"
 const char*   service,  // e.g. "ftp"
 const struct addrinfo* hints,
 struct addrinfo**      resultPtr);
// Also:  getnameinfo()
```

Gets info on host given integers

- Sets `resultPtr` to datastructure with info on host `hostName`
  - `hostName/service` effective tell `host:port`
  - returns integer: 0 == success, 0 != error.

Another strange struct!

```
struct addrinfo
{
  int              ai_flags;    // Used in hints
  int              ai_family;   // AF_INET, AF_INET6
                                // or AF_UNSPEC for
                                // IPv4,IPv6 or both
  int              ai_socktype; // SOCK_STREAM
                                //  or SOCK_DGRAM
  int              ai_protocol; // 0 = any protocol
  socklen_t        ai_addrlen;  // Len of next field
  struct sockaddr *ai_addr;     // (See prev slide)
  char            *ai_canonname;// Official hostname
  struct addrinfo *ai_next;     // For linked list
};
```

getaddrinfo() (derived from wikipedia example)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <netdb.h>
#include <netinet/in.h>
#include <sys/socket.h>

#ifndef   NI_MAXHOST
#define   NI_MAXHOST 1025
#endif

#define    LINE_LEN        256

#define    SERVICE_ADDR_SEP_STR  "://"

#define    SERVICE_ADDR_SEP_STR_LEN  \
                 (sizeof(SERVICE_ADDR_SEP_STR)-1)
```

getaddrinfo() (derived from wikipedia example)

```
char*   enterUrlName(char* urlName,
           int   urlNameLen
          )
{
  printf("\n"
    "URL "
    "(e.g. ftp://ctilinux1.cstcis.cti.depaul.edu)"
    " or a blank line to stop\n"
    ": "
   );
  fgets(urlName,urlNameLen,stdin);

  char*   cPtr   = strchr(urlName,'\n');

  if  (cPtr != NULL)
    *cPtr = '\0';

  return(urlName);
}
```

getaddrinfo() (derived from wikipedia example)

```
void   parse     (char*    serviceName,
           int      serviceNameLen,
           char*    addrName,
           int      addrNameLen,
           const char*  urlName
          )
{
  const char* cPtr;
  for  (cPtr = urlName;  isspace(*cPtr);  cPtr++);

  const char* sepPtr = strstr(cPtr,SERVICE_ADDR_SEP_STR);
  if  (sepPtr == NULL)
  {
    strncpy(serviceName,"",serviceNameLen);
    strncpy(addrName,cPtr,addrNameLen);
  }
  else
  {
    int   numServiceChars  = sepPtr-cPtr;
    strncpy(serviceName,cPtr,numServiceChars);
    serviceName[numServiceChars]  = '\0';
    strncpy(addrName,sepPtr+SERVICE_ADDR_SEP_STR_LEN,addrNameLen);
  }
}
```

getaddrinfo() (derived from wikipedia example)

```
void  describe  (const char*  serviceName,
           const char*  addrName
          )
{
  struct addrinfo* hostPtr;
  struct addrinfo* run;
  int  status = getaddrinfo
          (addrName,
           (serviceName[0] == '\0')
           ? NULL
           : serviceName,
           NULL,
           &hostPtr
          );

  if  (status != 0)
  {
    fprintf(stderr,gai_strerror(status));
    return;
  }
```

getaddrinfo() (derived from wikipedia example)

```
for  (run = hostPtr;  run != NULL;  run = run->ai_next)
{
  char hostname[NI_MAXHOST] = "";

  int   error = getnameinfo
                  (run->ai_addr,
                   run->ai_addrlen,
                   hostname, NI_MAXHOST, NULL, 0, 0);

  if (error != 0)
  {
    fprintf(stderr, "error in getnameinfo: %s\n",
            gai_strerror(error)
           );
    continue;
  }

  if (*hostname == '\0')
    printf("%-32s:",run->ai_canonname);
  else
    printf("%-32s:", hostname);
```

getaddrinfo() (derived from wikipedia example)

```
     switch  (run->ai_family)
    {
    case AF_INET :    printf("  (IPv4,"); break;
    case AF_INET6 :   printf("  (IPv6,"); break;
    case AF_UNSPEC :  printf("  (IPv4 & IPv6,"); break;
    case AF_UNIX :    printf("  (local Unix,");  break;
    case AF_IPX :     printf("  (Novell,");      break;
    case AF_APPLETALK:printf("  (Appletalk,");  break;
    case AF_PACKET:   printf("  (Lo-level packet,"); break;
    default :         printf("  (Unknown family?,");
    }

    switch  (run->ai_socktype)
    {
    case SOCK_STREAM :printf(" TCP)\n");    break;
    case SOCK_DGRAM : printf(" UDP)\n");    break;
    case SOCK_SEQPACKET:printf(" sequenced, reliable)\n"); break;
    case SOCK_RAW :   printf(" raw network protocol)\n");  break;
    case SOCK_RDM :   printf(" reliable w/o ordering)\n"); break;
    default :         printf(" unknown protocol?)\n");
    }
  }
}
```

getaddrinfo() (derived from wikipedia example)

```
int    main      ()
{
  char urlName[LINE_LEN];

  while  ( *enterUrlName(urlName,LINE_LEN) != '\0'  )
  {
    char  serviceName[LINE_LEN];
    char  addressName[LINE_LEN];

    parse(serviceName,LINE_LEN,addressName,LINE_LEN,urlName);
    describe(serviceName,addressName);
  }

  return(EXIT_SUCCESS);
}
```

connect()

```
  // Connect to server
  sockaddr_in server;

  // Clear server datastruct
  memset(&server, 0, sizeof(server));
  // Use TCP/IP
  server.sin_family      = AF_INET;
  // Tell port # in proper network byte order
  server.sin_port     = htons(portNumber);
  // Copy connectivity info from info on server ("hostPtr")
  server.sin_addr.s_addr =
     ((struct sockaddr_in*)hostPtr->ai_addr)->sin_addr.s_addr;

int status = connect(socketDescriptor,&server,sizeof(server));
```

- -1 means error

read(), write() and close()

As previously stated:

```
// Read from file/socket
// numRead==0 means "EndOfFile", numRead==-1 means "error"
int numRead =
    read(connectDescriptor,bufferAddress,bufferLen);
int numRead =
    recv(connectDescriptor,bufferAddress,bufferLen,int flags);

// Write to file/socket: numWritten == -1 means "error"
int numWritten =
    write(connectDescriptor,bufferAddress,bufferLen);
int numWritten =
    send(connectDescriptor,bufferAddress,bufferLen,int flags);

// Close connection: status == -1 means "error"
int status =  close(descriptor);
```

But sometimes you don't want to wait for socket input

```
int recv(int connectDescriptor,void*
  bufferPtr, int bufferLen, int flags)
```

Reads up to `bufferLen` bytes into the buffer pointed to by `bufferPtr` from file descriptor `connectDescriptor`. `flags` tells how to read, where `MSG_DONTWAIT` means "non-blocking".

Returns number of bytes read, or returns `-1` and sets global var `errno` to `EAGAIN` if the flag was `MSG_DONTWAIT` and there was nothing to read.

Short counts with `recv()`

- Short counts occur during:
  - Encounter End of file (EOF) when reading file (expected)
  - Reading text from terminal (also expected)
  - Reading from network or pipes if get interrupted by catching any sort of signal (an annoyance!)
  - ***Question:*** Did the fact that `read()` or `recv()` returned something mean that it ***got*** something, or that it ***was interrupted***?
- ***Oh no! Can nothing save us?!?!***

Robust I/O Package to the rescue!

```
/* From authors' thread-safe, buffered I/O package.
Same interface as read() */
ssize_t rio_readn(int fd, void *usrbuf, size_t n)
{
    size_t  nleft = n;
    ssize_t nread;
    char*   bufp = usrbuf;

    while (nleft > 0) {
      if ((nread = read(fd, bufp, nleft)) < 0) {
        if (errno==EINTR) /* Interrupted by sig handlr rtn? */
          nread = 0;      /* Yes: Call read() again */
        else
          return -1;      /* No:  Have some other error */
      }
      else if (nread == 0)/* Have EOF? */
        break;            /* Yes: Just quit loop */
      nleft -= nread;     /* Else that many fewer chars to get*/
      bufp  += nread;     /* Advance in buffer to read more */
    }
    return (n - nleft);   /* For non errors return val >= 0 */
}
```

Your turn!

- Write a client program that:
  1. Connects with the server
  2. Asks the user for text
  3. Sends the text to the server
  4. Gets the response back from the server and prints it

Next time

ncurses cursor control