71 Implement programs that simulate common Linux commands such as cat, ls, cp, mv, head, etc

```
FUNCTION main(ARGUMENTS):
    // ARGUMENTS is the array of input strings (e.g., ["./sim", "cat", "file.txt"])

    IF ARGUMENTS count < 2 THEN
        OUTPUT "Usage: program <command> [arguments...]"
        RETURN 1 // Indicate failure
    END IF

    COMMAND = ARGUMENTS[1] // The command name (e.g., "cat", "ls")

    IF COMMAND IS "cat" THEN
        CALL my_cat(ARGUMENTS)
    ELSE IF COMMAND IS "ls" THEN
        CALL my_ls(ARGUMENTS)
    ELSE IF COMMAND IS "cp" THEN
        CALL my_cp(ARGUMENTS)
    ELSE IF COMMAND IS "mv" THEN
        CALL my_mv(ARGUMENTS)
    ELSE IF COMMAND IS "head" THEN
        CALL my_head(ARGUMENTS)
    ELSE
        OUTPUT ERROR "Unknown command: " + COMMAND
        RETURN 1
    END IF

    RETURN 0 // Indicate success
END FUNCTION
```

2. `my_cat` (Concatenate and Display)

Simulates reading a file sequentially and writing its contents to standard output.

```
FUNCTION my_cat(ARGUMENTS):
    // Check for required argument: filename
    IF ARGUMENTS count < 3 THEN
        OUTPUT ERROR "Usage: cat <filename>"
        RETURN 1
    END IF

    FILENAME = ARGUMENTS[2]
```

```
    FILE_HANDLE = OPEN FILENAME in READ_MODE

    IF FILE_HANDLE IS NULL THEN
        OUTPUT ERROR "Failed to open file " + FILENAME
        RETURN 1
    END IF

    // Read and print content loop (byte by byte)
    LOOP:
        CHARACTER = READ single byte/character from FILE_HANDLE
        IF CHARACTER IS END_OF_FILE THEN
            BREAK LOOP
        END IF
        WRITE CHARACTER to STANDARD_OUTPUT
    END LOOP

    CLOSE FILE_HANDLE
    RETURN 0
END FUNCTION
```

3. `my_ls` (List Directory Contents)

Simulates iterating over entries in a specified directory.

```
FUNCTION my_ls(ARGUMENTS):
    // Determine the directory path (default to current directory ".")
    IF ARGUMENTS count < 3 THEN
        DIRECTORY_PATH = "."
    ELSE
        DIRECTORY_PATH = ARGUMENTS[2]
    END IF

    DIR_HANDLE = OPEN DIRECTORY_PATH

    IF DIR_HANDLE IS NULL THEN
        OUTPUT ERROR "Failed to open directory " + DIRECTORY_PATH
        RETURN 1
    END IF

    OUTPUT "Contents of " + DIRECTORY_PATH + ":"

    // Read directory entries until the end of the directory stream
    LOOP:
```

```
        ENTRY = READ next directory entry from DIR_HANDLE
        IF ENTRY IS NULL THEN
            BREAK LOOP
        END IF

        FILE_NAME = ENTRY's name

        // Filter out special directory entries: current (.) and parent (..)
        IF FILE_NAME IS NOT "." AND FILE_NAME IS NOT ".." THEN
            OUTPUT FILE_NAME
        END IF
    END LOOP

    CLOSE DIR_HANDLE
    RETURN 0
END FUNCTION
```

4. `my_cp` (Copy Files)

Simulates transferring data from a source file to a newly created destination file.

```
FUNCTION my_cp(ARGUMENTS):
    // Check for required arguments: source and destination
    IF ARGUMENTS count < 4 THEN
        OUTPUT ERROR "Usage: cp <source> <destination>"
        RETURN 1
    END IF

    SOURCE_NAME = ARGUMENTS[2]
    DEST_NAME = ARGUMENTS[3]

    SOURCE_HANDLE = OPEN SOURCE_NAME in READ_MODE
    IF SOURCE_HANDLE IS NULL THEN
        OUTPUT ERROR "Failed to open source file"
        RETURN 1
    END IF

    DEST_HANDLE = OPEN DEST_NAME in WRITE_MODE // Creates or overwrites
    IF DEST_HANDLE IS NULL THEN
        OUTPUT ERROR "Failed to open destination file"
        CLOSE SOURCE_HANDLE
        RETURN 1
    END IF
```

```
    // Copy loop (byte by byte)
    LOOP:
        CHARACTER = READ single byte/character from SOURCE_HANDLE
        IF CHARACTER IS END_OF_FILE THEN
            BREAK LOOP
        END IF
        WRITE CHARACTER to DEST_HANDLE
    END LOOP

    // Check for read/write errors during the loop
    IF ERROR OCCURRED during file operations THEN
        OUTPUT ERROR "Data transfer error during copy."
        RETURN 1
    END IF

    CLOSE SOURCE_HANDLE
    CLOSE DEST_HANDLE

    OUTPUT "Copied " + SOURCE_NAME + " to " + DEST_NAME
    RETURN 0
END FUNCTION
```

5. `my_mv` (Move/Rename Files)

Simulates moving or renaming a file using the underlying system rename functionality.

```
FUNCTION my_mv(ARGUMENTS):
    // Check for required arguments: oldname and newname
    IF ARGUMENTS count < 4 THEN
        OUTPUT ERROR "Usage: mv <oldname> <newname>"
        RETURN 1
    END IF

    OLD_NAME = ARGUMENTS[2]
    NEW_NAME = ARGUMENTS[3]

    // Use the operating system's rename function
    RESULT = SYSTEM_CALL_RENAME(OLD_NAME, NEW_NAME)

    IF RESULT IS SUCCESS THEN
        OUTPUT "Moved/Renamed " + OLD_NAME + " to " + NEW_NAME
        RETURN 0
```

```
    ELSE
        OUTPUT ERROR "Failed to move/rename file"
        RETURN 1
    END IF
END FUNCTION
```

6. `my_head` (Print Top Lines of a File)

Simulates reading and printing the first 10 lines of a file.

```
FUNCTION my_head(ARGUMENTS):
    // Check for required argument: filename
    IF ARGUMENTS count < 3 THEN
        OUTPUT ERROR "Usage: head <filename>"
        RETURN 1
    END IF

    FILENAME = ARGUMENTS[2]
    MAX_LINES_TO_PRINT = 10
    LINE_COUNT = 0

    FILE_HANDLE = OPEN FILENAME in READ_MODE

    IF FILE_HANDLE IS NULL THEN
        OUTPUT ERROR "Failed to open file " + FILENAME
        RETURN 1
    END IF

    // Read and print loop
    LOOP:
        IF LINE_COUNT >= MAX_LINES_TO_PRINT THEN
            BREAK LOOP
        END IF

        // Read the entire next line of text
        LINE = READ next full line from FILE_HANDLE (up to a newline or EOF)

        IF LINE IS NULL OR LINE IS END_OF_FILE THEN
            BREAK LOOP
        END IF

        WRITE LINE to STANDARD_OUTPUT
        LINE_COUNT = LINE_COUNT + 1
```

```
    END LOOP

    CLOSE FILE_HANDLE
    RETURN 0
END FUNCTION
```

2. `cat` (Concatenate and Display)

```
FUNCTION my_cat(ARGUMENTS):
    // Check for correct number of arguments: must be at least 3 (program name, "cat", filename)
    IF ARGUMENTS count < 3 THEN
        OUTPUT ERROR "Usage: cat <filename>"
        RETURN 1
    END IF

    FILENAME = ARGUMENTS[2]
    FILE_HANDLE = OPEN FILENAME in READ_MODE

    IF FILE_HANDLE IS NULL THEN
        OUTPUT ERROR "Failed to open file " + FILENAME
        RETURN 1
    END IF

    // Read and print loop
    LOOP:
        CHARACTER = READ single byte/character from FILE_HANDLE
        IF CHARACTER IS END_OF_FILE THEN
            BREAK LOOP
        END IF
        WRITE CHARACTER to STANDARD_OUTPUT
    END LOOP

    CLOSE FILE_HANDLE
    RETURN 0
END FUNCTION
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dirent.h>
#include <sys/stat.h>
#include <unistd.h>
```

```
void cat_command(char *filename) {
    FILE *file = fopen(filename, "r");
    if (!file) {
        perror("cat");
        return;
    }
    char ch;
    while ((ch = fgetc(file)) != EOF)
        putchar(ch);
    fclose(file);
}

void ls_command(char *path) {
    DIR *dir;
    struct dirent *entry;

    if (path == NULL)
        path = ".";

    dir = opendir(path);
    if (!dir) {
        perror("ls");
        return;
    }

    while ((entry = readdir(dir)) != NULL) {
        printf("%s\n", entry->d_name);
    }

    closedir(dir);
}

void cp_command(char *src, char *dest) {
    FILE *source = fopen(src, "rb");
    FILE *destination = fopen(dest, "wb");

    if (!source || !destination) {
        perror("cp");
        if (source) fclose(source);
        if (destination) fclose(destination);
        return;
    }

    char buffer[1024];
```

```
    size_t bytes;
    while ((bytes = fread(buffer, 1, sizeof(buffer), source)) > 0) {
        fwrite(buffer, 1, bytes, destination);
    }

    fclose(source);
    fclose(destination);
}

void mv_command(char *src, char *dest) {
    if (rename(src, dest) != 0) {
        perror("mv");
    }
}

void head_command(char *filename, int n) {
    FILE *file = fopen(filename, "r");
    if (!file) {
        perror("head");
        return;
    }

    char line[1024];
    int count = 0;

    while (fgets(line, sizeof(line), file) && count < n) {
        printf("%s", line);
        count++;
    }

    fclose(file);
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Usage:\n");
        printf("  %s cat <file>\n", argv[0]);
        printf("  %s ls [directory]\n", argv[0]);
        printf("  %s cp <source> <destination>\n", argv[0]);
        printf("  %s mv <source> <destination>\n", argv[0]);
        printf("  %s head <file> <num_lines>\n", argv[0]);
        return 1;
    }
```

```
    if (strcmp(argv[1], "cat") == 0) {
        if (argc < 3) printf("cat: missing file operand\n");
        else cat_command(argv[2]);
    }
    else if (strcmp(argv[1], "ls") == 0) {
        if (argc < 3) ls_command(".");
        else ls_command(argv[2]);
    }
    else if (strcmp(argv[1], "cp") == 0) {
        if (argc < 4) printf("cp: missing file operand\n");
        else cp_command(argv[2], argv[3]);
    }
    else if (strcmp(argv[1], "mv") == 0) {
        if (argc < 4) printf("mv: missing file operand\n");
        else mv_command(argv[2], argv[3]);
    }
    else if (strcmp(argv[1], "head") == 0) {
        if (argc < 4) printf("head: missing arguments\n");
        else head_command(argv[2], atoi(argv[3]));
    }
    else {
        printf("Unknown command: %s\n", argv[1]);
    }

    return 0;
}
```

72. Use semaphores to ensure `f1()` executes before `f2()` (prompt username before password)

PSEUDOCODE: ensure f1 before f2 using POSIX semaphore (thread version)
1. Initialize semaphore sem with initial value 0 (sem_init or sem_open for named)
2. Start thread t1 that runs f1:
    a. f1: prompt "Enter username: "
    b. read username from stdin
    c. optionally validate/store username
    d. sem_post(sem)   // signal that f1 is done
    e. exit thread
3. Start thread t2 that runs f2:
    a. sem_wait(sem)   // block until f1 posts
    b. prompt "Enter password: "

    c. read password from stdin
    d. exit thread
4. Join threads
5. Destroy semaphore
6. Exit

```
// File: sem_order.c
// Compile: gcc -o sem_order sem_order.c -pthread
#include <stdio.h>
#include <semaphore.h>
#include <pthread.h>
#include <stdlib.h>

sem_t s;

void *f1(void *arg){
    char user[128];
    printf("Enter username: ");
    if(!fgets(user, sizeof(user), stdin)) return NULL;
    printf("Got username: %s", user);
    sem_post(&s); // signal f2
    return NULL;
}
void *f2(void *arg){
    sem_wait(&s); // wait until f1 posts
    char pass[128];
    printf("Enter password: ");
    if(!fgets(pass, sizeof(pass), stdin)) return NULL;
    printf("Got password: %s", pass);
    return NULL;
}
int main(){
    pthread_t t1,t2;
    sem_init(&s, 0, 0);
    pthread_create(&t1,NULL,f1,NULL);
    pthread_create(&t2,NULL,f2,NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
    sem_destroy(&s);
    return 0;
}
```

73-76 Two programs exchanging messages with System V message queues (Process1 ↔ Process2): dialog Hi? → Hello → I am fine

PSEUDOCODE: Proc1 (mq_proc1)
1. key = ftok(".", 'M')
2. qid = msgget(key, IPC_CREAT | 0666)
3. // send message type 1
   msg.type = 1
   msg.text = "Hi?"
   msgsnd(qid, &msg, strlen(msg.text)+1, 0)
4. // wait for reply of type 2
   msgrcv(qid, &msg, sizeof(msg.text), 2, 0)
   print "Proc1 got: " + msg.text
5. // send message type 1 again: "I am fine"
   msg.type = 1
   msg.text = "I am fine"
   msgsnd(qid, &msg, strlen(msg.text)+1, 0)
6. Exit (optionally do not remove queue)

PSEUDOCODE: Proc2 (mq_proc2)
1. key = ftok(".", 'M')
2. qid = msgget(key, IPC_CREAT | 0666)
3. // receive message type 1
   msgrcv(qid, &msg, sizeof(msg.text), 1, 0)
   print "Proc2 got: " + msg.text
4. // send reply type 2: "Hello"
   msg.type = 2
   msg.text = "Hello"
   msgsnd(qid, &msg, strlen(msg.text)+1, 0)
5. // receive next type 1 ("I am fine")
   msgrcv(qid, &msg, sizeof(msg.text), 1, 0)
   print "Proc2 got: " + msg.text
6. msgctl(qid, IPC_RMID, NULL) // optionally remove queue
7. Exit

```
// File: mq_proc1.c
// Compile: gcc -o mq_proc1 mq_proc1.c
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>

struct msgbuf { long mtype; char mtext[128]; };
```

```
int main(){
    key_t key = ftok(".", 'M');
    int qid = msgget(key, IPC_CREAT | 0666);
    struct msgbuf msg;
    // send "Hi?"
    msg.mtype = 1; strcpy(msg.mtext, "Hi?");
    msgsnd(qid, &msg, strlen(msg.mtext)+1, 0);
    // receive reply (type 2)
    msgrcv(qid, &msg, sizeof(msg.mtext), 2, 0);
    printf("Proc1 got: %s\n", msg.mtext);
    // send "I am fine"
    msg.mtype = 1; strcpy(msg.mtext, "I am fine");
    msgsnd(qid, &msg, strlen(msg.mtext)+1, 0);
    return 0;
}

// File: mq_proc2.c
// Compile: gcc -o mq_proc2 mq_proc2.c
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>

struct msgbuf { long mtype; char mtext[128]; };

int main(){
    key_t key = ftok(".", 'M');
    int qid = msgget(key, IPC_CREAT | 0666);
    struct msgbuf msg;
    // receive Hi? (type 1)
    msgrcv(qid, &msg, sizeof(msg.mtext), 1, 0);
    printf("Proc2 got: %s\n", msg.mtext);
    // reply "Hello" as type 2
    msg.mtype = 2; strcpy(msg.mtext, "Hello");
    msgsnd(qid, &msg, strlen(msg.mtext)+1, 0);
    // receive "I am fine" (type 1)
    msgrcv(qid, &msg, sizeof(msg.mtext), 1, 0);
    printf("Proc2 got: %s\n", msg.mtext);
    // remove queue
    msgctl(qid, IPC_RMID, NULL);
    return 0;
}
```

77. TCP program demonstrating socket system calls (Client + Server) — iterative/simple

PSEUDOCODE: TCP server
1. sockfd = socket(AF_INET, SOCK_STREAM, 0)
2. bind address: { family=AF_INET, port=PORT, addr=INADDR_ANY } -> bind(sockfd, &addr)
3. listen(sockfd, backlog)
4. Loop:
    a. connfd = accept(sockfd, client_addr, &addrlen)   // blocking
    b. handle client:
        i. read data from connfd into buffer (read/recv)
        ii. process or print data
        iii. optionally send response via write/send
        iv. close(connfd)
5. Close(sockfd)

PSEUDOCODE: TCP client
1. sockfd = socket(AF_INET, SOCK_STREAM, 0)
2. set server address { family=AF_INET, port=PORT, addr=server_ip }
3. connect(sockfd, &server_addr)
4. write/send data to server
5. read/recv response from server
6. close(sockfd)

```
// File: tcp_server.c
// Compile: gcc -o tcp_server tcp_server.c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

int main(){
   int sock = socket(AF_INET, SOCK_STREAM, 0);
   struct sockaddr_in addr = {0};
   addr.sin_family = AF_INET; addr.sin_port = htons(9000); addr.sin_addr.s_addr =
INADDR_ANY;
   bind(sock, (struct sockaddr*)&addr, sizeof(addr));
   listen(sock, 5);
   printf("TCP server listening on 9000\n");
   int conn = accept(sock, NULL, NULL);
   char buf[512];
   int n = read(conn, buf, sizeof(buf)-1);
```

```
    if(n>0){ buf[n]=0; printf("Server got: %s\n", buf); write(conn, "ACK", 3); }
    close(conn); close(sock);
    return 0;
}

// File: tcp_client.c
// Compile: gcc -o tcp_client tcp_client.c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

int main(){
    int s = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in addr = {0};
    addr.sin_family = AF_INET; addr.sin_port = htons(9000);
    inet_pton(AF_INET, "127.0.0.1", &addr.sin_addr);
    connect(s, (struct sockaddr*)&addr, sizeof(addr));
    write(s, "Hello from client", 17);
    char buf[64]; int n = read(s, buf, sizeof(buf)-1);
    if(n>0){ buf[n]=0; printf("Client got: %s\n", buf); }
    close(s); return 0;
}
```

78. UDP program demonstrating socket system calls (Client + Server)

```
PSEUDOCODE: UDP server
1. sockfd = socket(AF_INET, SOCK_DGRAM, 0)
2. bind(sockfd, server_addr)   // server listens on port
3. Loop:
     a. recvfrom(sockfd, buffer, bufsize, 0, client_addr, &addrlen)
     b. optionally print/process buffer
     c. sendto(sockfd, reply, len, 0, client_addr, addrlen)
4. close(sockfd)


PSEUDOCODE: UDP client
1. sockfd = socket(AF_INET, SOCK_DGRAM, 0)
2. server_addr = { ip, port }
3. sendto(sockfd, message, len, 0, server_addr, sizeof)
4. recvfrom(sockfd, buffer, bufsize, 0, &from_addr, &from_len)
```

5. close(sockfd)

```
// File: udp_server.c
// Compile: gcc -o udp_server udp_server.c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

int main(){
   int s = socket(AF_INET, SOCK_DGRAM, 0);
   struct sockaddr_in addr = {0}, cli={0};
   addr.sin_family=AF_INET; addr.sin_port=htons(9001); addr.sin_addr.s_addr=INADDR_ANY;
   bind(s, (struct sockaddr*)&addr, sizeof(addr));
   char buf[256]; socklen_t l=sizeof(cli);
   int n = recvfrom(s, buf, sizeof(buf)-1, 0, (struct sockaddr*)&cli, &l);
   if(n>0){ buf[n]=0; printf("UDP server got: %s\n", buf); sendto(s, "PONG", 4, 0, (struct
sockaddr*)&cli, l); }
   close(s); return 0;
}

// File: udp_client.c
// Compile: gcc -o udp_client udp_client.c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

int main(){
   int s = socket(AF_INET, SOCK_DGRAM, 0);
   struct sockaddr_in addr = {0};
   addr.sin_family=AF_INET; addr.sin_port=htons(9001);
   inet_pton(AF_INET,"127.0.0.1",&addr.sin_addr);
   sendto(s, "PING", 4, 0, (struct sockaddr*)&addr, sizeof(addr));
   char buf[64]; socklen_t l=sizeof(addr);
   int n = recvfrom(s, buf, sizeof(buf)-1, 0, (struct sockaddr*)&addr, &l);
   if(n>0){ buf[n]=0; printf("UDP client got: %s\n", buf); }
   close(s); return 0;
}
```

79. Echo server over TCP (iterative and/or concurrent)

PSEUDOCODE: iterative TCP echo server
1. create socket, bind, listen (same as TCP server)
2. while true:
    a. connfd = accept(...)
    b. while (n = read(connfd, buf, BUFSIZE)) > 0:
        write(connfd, buf, n)   // echo back
    c. close(connfd)
3. close(listenfd)

```
// File: tcp_echo_server.c
// Compile: gcc -o tcp_echo_server tcp_echo_server.c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

int main(){
   int s = socket(AF_INET, SOCK_STREAM, 0);
   struct sockaddr_in a={0}; a.sin_family=AF_INET; a.sin_port=htons(9100);
a.sin_addr.s_addr=INADDR_ANY;
   bind(s,(struct sockaddr*)&a,sizeof(a)); listen(s,5);
   printf("Echo server on 9100\n");
   while(1){
      int c = accept(s,NULL,NULL);
      char buf[512]; ssize_t n;
      while((n=read(c,buf,sizeof(buf)))>0){ write(c, buf, n); }
      close(c);
   }
   close(s); return 0;
}
```

PSEUDOCODE: concurrent TCP echo server (fork)
1. create socket, bind, listen
2. while true:
    a. connfd = accept(...)
    b. pid = fork()
    c. if pid == 0 (child):
        close(listenfd)
        echo loop: read->write on connfd until EOF
        close(connfd); exit child
      else (parent):
        close(connfd)

optionally reap child processes (waitpid with WNOHANG)

80. Echo server over UDP (iterative / concurrent)

PSEUDOCODE: UDP echo server (iterative)
1. sockfd = socket(AF_INET, SOCK_DGRAM, 0)
2. bind(sockfd, server_addr)
3. Loop forever:
    a. n = recvfrom(sockfd, buf, BUFSIZE, 0, &client_addr, &addrlen)
    b. sendto(sockfd, buf, n, 0, &client_addr, addrlen)   // echo back
4. close(sockfd)

```
// File: udp_echo_server.c
// Compile: gcc -o udp_echo_server udp_echo_server.c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

int main(){
   int s = socket(AF_INET, SOCK_DGRAM, 0);
   struct sockaddr_in a={0}, cli={0};
   a.sin_family=AF_INET; a.sin_port=htons(9200); a.sin_addr.s_addr=INADDR_ANY;
   bind(s,(struct sockaddr*)&a,sizeof(a));
   char buf[1024]; socklen_t l=sizeof(cli);
   while(1){
      ssize_t n = recvfrom(s, buf, sizeof(buf), 0, (struct sockaddr*)&cli, &l);
      if(n>0) sendto(s, buf, n, 0, (struct sockaddr*)&cli, l);
   }
   close(s); return 0;
}
```

81. Unnamed pipe: send data from parent to child

PSEUDOCODE: parent->child via unnamed pipe
1. int fd[2]; pipe(fd)   // fd[0] read end, fd[1] write end
2. pid = fork()
3. if pid == 0 (child):
    a. close(fd[1])   // close write end
    b. loop: n = read(fd[0], buf, BUFSIZE); if n<=0 break; write(STDOUT, buf, n)

    c. close(fd[0]); exit
4. else (parent):
    a. close(fd[0])   // close read end
    b. write(fd[1], data, len) // send data
    c. close(fd[1])
    d. Optionally waitpid(pid)
5. Exit

```
// File: pipe_parent_child.c
// Compile: gcc -o pipe_parent_child pipe_parent_child.c
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

int main(){
   int fd[2];
   if(pipe(fd)==-1){ perror("pipe"); return 1; }
   pid_t pid = fork();
   if(pid<0){ perror("fork"); return 1; }
   if(pid==0){
      close(fd[1]);
      char buf[256]; int n = read(fd[0], buf, sizeof(buf)-1);
      if(n>0){ buf[n]=0; printf("Child received: %s\n", buf); }
      close(fd[0]); return 0;
   } else {
      close(fd[0]);
      const char *msg = "Message from parent";
      write(fd[1], msg, strlen(msg));
      close(fd[1]);
      wait(NULL);
   }
   return 0;
}
```

82. Unnamed pipe: send a file from parent to child

PSEUDOCODE: send file contents parent -> child via pipe
1. Check argv for filename; open file for read (fd_file)
2. pipe(fd)
3. pid = fork()

4. if pid == 0 (child):
    a. close(fd[1])
    b. loop: n = read(fd[0], buf, BUFSIZE); if n<=0 break; write(STDOUT, buf, n)
    c. close(fd[0]); exit
5. else (parent):
    a. close(fd[0])
    b. while (n = read(fd_file, buf, BUFSIZE)) > 0:
        write(fd[1], buf, n)
    c. close(fd_file); close(fd[1])
    d. waitpid(pid)
6. Exit

```
// File: pipe_file_transfer.c
// Compile: gcc -o pipe_file_transfer pipe_file_transfer.c
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/wait.h>

int main(int argc,char **argv){
   if(argc<2){ printf("Usage: %s filename\n", argv[0]); return 1; }
   int fd[2]; if(pipe(fd)==-1){ perror("pipe"); return 1; }
   pid_t pid = fork();
   if(pid<0){ perror("fork"); return 1; }
   if(pid==0){
      close(fd[1]);
      char buf[4096]; int n;
      while((n=read(fd[0], buf, sizeof(buf)))>0) write(1, buf, n);
      close(fd[0]); return 0;
   } else {
      close(fd[0]);
      int f = open(argv[1], O_RDONLY);
      if(f<0){ perror("open"); close(fd[1]); return 1; }
      char buf[4096]; int n;
      while((n=read(f, buf, sizeof(buf)))>0) write(fd[1], buf, n);
      close(f); close(fd[1]);
      wait(NULL);
   }
   return 0;
}
```

83. Pipe acting as a filter: convert uppercase text to lowercase (parent writes, child converts)

PSEUDOCODE: filter uppercase->lowercase via pipe
1. pipe(fd)
2. pid = fork()
3. if pid == 0 (child - filter):
    a. close(fd[1])
    b. while (n = read(fd[0], buf, BUFSIZE)) > 0:
        for i in 0..n-1: buf[i] = tolower(buf[i])
        write(STDOUT, buf, n)   // or to another fd
    c. close(fd[0]); exit
4. else (parent - producer):
    a. close(fd[0])
    b. If input filename provided:
        open file, read chunks, write(fd[1], chunk, n)
      else:
        read from STDIN and write to fd[1]
    c. close(fd[1]); waitpid(child)
5. Exit

```
// File: pipe_filter_tolower.c
// Compile: gcc -o pipe_filter_tolower pipe_filter_tolower.c
#include <stdio.h>
#include <unistd.h>
#include <ctype.h>
#include <string.h>
#include <sys/wait.h>

int main(int argc,char **argv){
   int fd[2]; if(pipe(fd)==-1){ perror("pipe"); return 1; }
   pid_t pid = fork();
   if(pid<0){ perror("fork"); return 1; }
   if(pid==0){
      close(fd[1]);
      char buf[1024]; int n;
      while((n=read(fd[0], buf, sizeof(buf)))>0){
         for(int i=0;i<n;i++) buf[i]=tolower((unsigned char)buf[i]);
         write(1, buf, n);
      }
      close(fd[0]); return 0;
   } else {
      close(fd[0]);
      if(argc>1){
```

```
        FILE *f = fopen(argv[1],"r");
        char line[1024];
        while(fgets(line,sizeof(line),f)) write(fd[1], line, strlen(line));
        fclose(f);
    } else {
        char buf[1024];
        while(fgets(buf,sizeof(buf),stdin)) write(fd[1], buf, strlen(buf));
    }
    close(fd[1]);
    wait(NULL);
  }
  return 0;
}
```

84. Semaphore usage with `fork()` so two processes run simultaneously and coordinate via semaphores; short note on `sys/sem.h` (SysV) vs `semaphore.h` (POSIX)

PSEUDOCODE: sem_fork example (POSIX named semaphore)
1. sem = sem_open("/mysem", O_CREAT | O_EXCL, 0666, initial_value)
2. pid = fork()
3. if pid == 0 (child):
    a. sem_wait(sem)   // wait for parent to post
    b. print "child proceeding"
    c. sem_close(sem); exit
4. else (parent):
    a. do some work
    b. sem_post(sem)   // signal child
    c. wait for child; sem_unlink("/mysem"); sem_close(sem)
5. Exit

```
// File: sem_fork.c
// Compile: gcc -o sem_fork sem_fork.c -pthread -lrt
#include <stdio.h>
#include <semaphore.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/wait.h>

int main(){
  sem_t *s = sem_open("/mysem_demo", O_CREAT | O_EXCL, 0666, 0);
  if(s == SEM_FAILED){ perror("sem_open"); return 1; }
```

```
    pid_t pid = fork();
    if(pid<0){ perror("fork"); sem_unlink("/mysem_demo"); return 1; }
    if(pid==0){
        sem_wait(s);
        printf("Child: after wait\n");
        sem_close(s);
        return 0;
    } else {
        printf("Parent: doing work, then posting\n");
        sleep(1);
        sem_post(s);
        wait(NULL);
        sem_unlink("/mysem_demo");
    }
    return 0;
}
```

85. Three separate programs operating on the same named semaphore

PSEUDOCODE: Program A - initialize semaphore
1. sem = sem_open("/shared_sem_example", O_CREAT | O_EXCL, 0666, initial_value)
2. print semaphore name or success
3. sem_close(sem)
4. Exit

PSEUDOCODE: Program B - perform P (wait)
1. sem = sem_open("/shared_sem_example", 0)   // open existing
2. print "waiting on semaphore"
3. sem_wait(sem)
4. print "inside critical section"
5. optionally wait for user input to hold it
6. sem_close(sem)
7. Exit

PSEUDOCODE: Program C - perform V (signal)
1. sem = sem_open("/shared_sem_example", 0)
2. sem_post(sem)
3. print "posted semaphore"
4. sem_close(sem)
5. Exit

```c
// File: sem_init.c
// Compile: gcc -o sem_init sem_init.c -lrt
#include <stdio.h>
#include <semaphore.h>
#include <fcntl.h>

int main(){
    sem_t *s = sem_open("/shared_sem_example", O_CREAT | O_EXCL, 0666, 1);
    if(s==SEM_FAILED){ perror("sem_open"); return 1; }
    printf("Semaphore created: /shared_sem_example\n");
    sem_close(s); return 0;
}


// File: sem_wait_prog.c
// Compile: gcc -o sem_wait_prog sem_wait_prog.c -lrt
#include <stdio.h>
#include <semaphore.h>
#include <fcntl.h>

int main(){
    sem_t *s = sem_open("/shared_sem_example", 0);
    if(s==SEM_FAILED){ perror("sem_open"); return 1; }
    printf("Waiting (P) on semaphore...\n");
    sem_wait(s);
    printf("Inside critical section (after wait). Press Enter to release.\n");
    getchar();
    sem_close(s); return 0;
}

// File: sem_post_prog.c
// Compile: gcc -o sem_post_prog sem_post_prog.c -lrt
#include <stdio.h>
#include <semaphore.h>
#include <fcntl.h>

int main(){
    sem_t *s = sem_open("/shared_sem_example", 0);
    if(s==SEM_FAILED){ perror("sem_open"); return 1; }
    sem_post(s);
    printf("Performed V (signal) on semaphore\n");
    sem_close(s); return 0;
}
```

86. File locking using `lockf()`

```
PSEUDOCODE: lockf_demo
1. fd = open("lockfile.txt", O_CREAT | O_RDWR, 0666)
2. print "trying to lock"
3. if lockf(fd, F_LOCK, 0) == 0:
     a. print "locked; press Enter to release"
     b. wait for user input
     c. lockf(fd, F_ULOCK, 0)
     d. print "unlocked"
   else:
     a. print lockf error
4. close(fd)
5. Exit
```

```
// File: lockf_demo.c
// Compile: gcc -o lockf_demo lockf_demo.c
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

int main(){
    int fd = open("lockfile.txt", O_CREAT | O_RDWR, 0666);
    if(fd<0){ perror("open"); return 1; }
    printf("Trying to place lock (blocking)...\n");
    if(lockf(fd, F_LOCK, 0) == 0){
        printf("Locked. Press Enter to unlock.\n");
        getchar();
        lockf(fd, F_ULOCK, 0);
        printf("Unlocked.\n");
    } else perror("lockf");
    close(fd); return 0;
}
```

87. File locking using flock()

```
PSEUDOCODE: flock_demo
1. fd = open("flockfile.txt", O_CREAT | O_RDWR, 0666)
2. print "Attempting flock exclusive (blocking)"
3. if flock(fd, LOCK_EX) == 0:
     a. print "locked; press Enter to unlock"
     b. wait for user input
     c. flock(fd, LOCK_UN)
     d. print "unlocked"
   else:
     a. print flock error
4. close(fd)
5. Exit
```

```
// File: flock_demo.c
// Compile: gcc -o flock_demo flock_demo.c
#include <stdio.h>
#include <sys/file.h>
#include <unistd.h>
#include <fcntl.h>

int main(){
    int fd = open("flockfile.txt", O_CREAT | O_RDWR, 0666);
    if(fd<0){ perror("open"); return 1; }
    printf("Attempting flock (exclusive, blocking)...\n");
    if(flock(fd, LOCK_EX) == 0){
        printf("Locked by flock. Press Enter to unlock.\n");
        getchar();
        flock(fd, LOCK_UN);
        printf("Unlocked.\n");
    } else perror("flock");
    close(fd); return 0;
}
```