Problem 1 Solution: File Encryption Program

Directory Structure

Create a directory named `v10/` with the following files:

- `main.c`
- `readspecs.c`
- `outputfilename.c`
- `keyconvert.c`
- `fileprocess.c`
- `Makefile`

Implementation

1. main.c

```
#include <stdio.h>
#include <stdlib.h>

void readspecs(char *, char *);
void outputfilename(char *);
void keyconvert(char *, int *);
void fileprocess(int, FILE *, FILE *);

extern char outputfile[25];

int main() {
    char seckey[8];
    char inputfile[21];
    int seckeyint = 0;

    readspecs(seckey, inputfile);
    outputfilename(inputfile);
    keyconvert(seckey, &seckeyint);

    FILE *infile = fopen(inputfile, "rb");
    if (infile == NULL) {
        fprintf(stderr, "Error opening input file\n");
        exit(1);
    }

    FILE *outfile = fopen(outputfile, "wb");
    if (outfile == NULL) {
        fprintf(stderr, "Error opening output file\n");
        fclose(infile);
```

```
        exit(1);
    }

    fileprocess(seckeyint, infile, outfile);

    fclose(infile);
    fclose(outfile);

    return 0;
}
```

2. readspecs.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

void readspecs(char *seckey, char *inputfile) {
    char input[30];
    if (fgets(input, sizeof(input), stdin) == NULL) {
        fprintf(stderr, "Error reading input\n");
        exit(1);
    }

    input[strcspn(input, "\n")] = 0;  // Remove newline

    if (strlen(input) < 9 || strlen(input) > 28) {
        fprintf(stderr, "Invalid input length\n");
        exit(1);
    }

    for (int i = 0; i < 8; i++) {
        if (input[i] != '0' && input[i] != '1') {
            fprintf(stderr, "Invalid secret key\n");
            exit(1);
        }
        seckey[i] = input[i];
    }

    strcpy(inputfile, input + 8);

    for (int i = 0; inputfile[i]; i++) {
        if (!isalnum(inputfile[i]) && inputfile[i] != '.') {
            fprintf(stderr, "Invalid filename\n");
            exit(1);
        }
    }
```

```
}
```

3. outputfilename.c

```
#include <string.h>

char outputfile[25];

void outputfilename(char *inputfile) {
    int len = strlen(inputfile);
    if (len >= 4 && strcmp(inputfile + len - 4, ".enc") == 0) {
        strncpy(outputfile, inputfile, len - 4);
        outputfile[len - 4] = '\0';
    } else {
        strcpy(outputfile, inputfile);
        strcat(outputfile, ".enc");
    }
}
```

4. keyconvert.c

```
void keyconvert(char *seckey, int *seckeyint) {
    /*
    Logic: We use a bitmask to convert the 8-byte secret key to an integer.
    We iterate through the 8 characters of the secret key.
    For each character, we left-shift the bitmask by 1 and OR it with 1 if
    Finally, we apply the bitmask to *seckeyint using bitwise AND.
    */
    unsigned char bitmask = 0;
    for (int i = 0; i < 8; i++) {
        bitmask <<= 1;
        bitmask |= (seckey[i] == '1');
    }
    *seckeyint = bitmask;
}
```

5. fileprocess.c

```
#include <stdio.h>

void fileprocess(int seckeyint, FILE *infile, FILE *outfile) {
    int ch;
    while ((ch = fgetc(infile)) != EOF) {
        ch ^= seckeyint;
        fputc(ch, outfile);
    }
}
```

6. Makefile

```
CC = gcc
CFLAGS = -Wall -Wextra -std=c99
TARGET = encryptafile
OBJS = main.o readspecs.o outputfilename.o keyconvert.o fileprocess.o

all: $(TARGET)

$(TARGET): $(OBJS)
    $(CC) $(CFLAGS) -o $@ $^

main.o: main.c
    $(CC) $(CFLAGS) -c main.c

readspecs.o: readspecs.c
    $(CC) $(CFLAGS) -c readspecs.c

outputfilename.o: outputfilename.c
    $(CC) $(CFLAGS) -c outputfilename.c

keyconvert.o: keyconvert.c
    $(CC) $(CFLAGS) -c keyconvert.c

fileprocess.o: fileprocess.c
    $(CC) $(CFLAGS) -c fileprocess.c

clean:
    rm -f $(OBJS) $(TARGET)
```

Compilation and Testing

To test the program:

```
echo "01001101test.txt" | ./encryptafile
```

To decrypt it, run:

```
echo "01001101test.txt.enc" | ./encryptafile
```

use the `diff` utility to compare the original and decrypted files:

```
diff test.txt test.txt.enc
```

Problem2

- `main.c`
- `numlines.c`

- `readlines.c`
- `revlines.c`
- `Makefile`

Implementation

1. main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int numlines(FILE *);
void readlines(FILE *, char **);
void revlines(FILE *, char **, int);

int main() {
    char filename[16];
    char outfilename[20];
    int line_count;
    char **inputlines;

    // Read filename from stdin
    if (fgets(filename, sizeof(filename), stdin) == NULL) {
        printf("Error reading filename\n");
        return 1;
    }
    filename[strcspn(filename, "\n")] = 0; // Remove newline

    if (strlen(filename) > 15) {
        printf("Filename too long\n");
        return 1;
    }

    // Open input file
    FILE *infile = fopen(filename, "r");
    if (infile == NULL) {
        printf("Error opening input file\n");
        return 1;
    }

    // Count number of lines
    line_count = numlines(infile);
    if (line_count == 0) {
        printf("Input file is empty\n");
        fclose(infile);
        return 1;
```

```
}

// Allocate memory for line pointers
inputlines = (char **)malloc(line_count * sizeof(char *));
if (inputlines == NULL) {
    printf("Memory allocation failed\n");
    fclose(infile);
    return 1;
}

// Reopen input file to read from beginning
fclose(infile);
infile = fopen(filename, "r");
if (infile == NULL) {
    printf("Error reopening input file\n");
    free(inputlines);
    return 1;
}

// Read lines into memory
readlines(infile, inputlines);
fclose(infile);

// Create output filename
strcpy(outfilename, filename);
strcat(outfilename, ".rev");

// Open output file
FILE *outfile = fopen(outfilename, "w");
if (outfile == NULL) {
    printf("Error opening output file\n");
    for (int i = 0; i < line_count; i++) {
        free(inputlines[i]);
    }
    free(inputlines);
    return 1;
}

// Reverse lines and write to output file
revlines(outfile, inputlines, line_count);
fclose(outfile);

// Free allocated memory
for (int i = 0; i < line_count; i++) {
    free(inputlines[i]);
}
free(inputlines);
```

```
    printf("File processed successfully\n");
    return 0;
}
```

2. numlines.c

```
#include <stdio.h>

int numlines(FILE *fp) {
    int count = 0;
    int ch;

    while ((ch = fgetc(fp)) != EOF) {
        if (ch == '\n') {
            count++;
        }
    }

    // If file is not empty and doesn't end with '\n', count the last line
    if (count > 0 || ftell(fp) > 0) {
        count++;
    }

    return count;
}
```

3. readlines.c

```
#include <stdio.h>
#include <stdlib.h>

void readlines(FILE *fp, char **lines) {
    char templine[26];
    int line_index = 0;
    int char_index = 0;

    while (1) {
        int ch = fgetc(fp);
        if (ch == EOF) {
            if (char_index > 0) {
                templine[char_index] = '\0';
                lines[line_index] = malloc(char_index + 1);
                for (int i = 0; i <= char_index; i++) {
                    lines[line_index][i] = templine[i];
                }
            }
            break;
```

```
        }

        templine[char_index++] = ch;

        if (ch == '\n' || char_index == 25) {
            templine[char_index] = '\0';
            lines[line_index] = malloc(char_index + 1);
            for (int i = 0; i <= char_index; i++) {
                lines[line_index][i] = templine[i];
            }
            line_index++;
            char_index = 0;
        }
    }
}
```

4. revlines.c

```
#include <stdio.h>
#include <string.h>

void revlines(FILE *fp, char **lines, int num_lines) {
    for (int i = num_lines - 1; i >= 0; i--) {
        fprintf(fp, "%s", lines[i]);
        if (i != 0) {
            fprintf(fp, "\n");
        }
    }
}
```

Makefile

```
CC = gcc
CFLAGS = -Wall -Wextra -std=c99
TARGET = reverselines
OBJS = main.o numlines.o readlines.o revlines.o

all: $(TARGET)

$(TARGET): $(OBJS)
    $(CC) $(CFLAGS) -o $@ $^

main.o: main.c
    $(CC) $(CFLAGS) -c main.c

numlines.o: numlines.c
    $(CC) $(CFLAGS) -c numlines.c
```

```
readlines.o: readlines.c
    $(CC) $(CFLAGS) -c readlines.c

revlines.o: revlines.c
    $(CC) $(CFLAGS) -c revlines.c

clean:
    rm -f $(OBJS) $(TARGET)
```

Compilation and Testing

To test the program:

1. Create a test input file (e.g., `test.txt`):

```
echo -e "This is a test file\nfor P3.\nReverse the lines." > test.txt
```

2. Run the program:

```
echo "test.txt" | ./reverselines
```

3. Check the output file `test.txt.rev`:

```
cat test.txt.rev
```

The output should show the lines in reverse order:

```
Reverse the lines.
for P3.
This is a test file
```

This implementation follows the specifications given in the problem statement, including proper memory allocation and deallocation, error handling, and file processing.

Bonus Problem

The `reverselines` application does not necessarily need to mandate that the input file is ASCII. It can work with binary files as well, with some considerations. Here's the reasoning:

1. **Line Delimitation:** The program uses newline characters ('\n') to determine the end of each line. This is the only ASCII-specific assumption in the program.
2. **Binary Data Handling:** The program reads and writes data as-is, without interpreting or modifying it (except for the newline handling). This means it can handle arbitrary binary data within each line.
3. **Null Character Handling:** In binary files, null characters ('\0') may appear within the data. The program should be able to handle these without truncating the lines prematurely.

To demonstrate that the program can handle binary files, we can modify it slightly and then test it with a binary file. Here are the necessary modifications:

Modified readlines.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void readlines(FILE *fp, char **lines) {
    char templine[26];
    int line_index = 0;
    size_t len;

    while (1) {
        len = 0;
        int c;
        while ((c = fgetc(fp)) != EOF && c != '\n' && len < 25) {
            templine[len++] = c;
        }
        templine[len] = '\0';

        // Allocate memory for the line
        lines[line_index] = (char *)malloc(len + 1);
        if (lines[line_index] == NULL) {
            fprintf(stderr, "Memory allocation failed\n");
            exit(1);
        }

        // Copy the line
        memcpy(lines[line_index], templine, len + 1);
        line_index++;

        if (c == EOF) break;
    }
}
```

Modified revlines.c

```
#include <stdio.h>
#include <string.h>

void revlines(FILE *fp, char **lines, int num_lines) {
    for (int i = num_lines - 1; i >= 0; i--) {
        fwrite(lines[i], 1, strlen(lines[i]), fp);
        if (i != 0) {
            fputc('\n', fp);
        }
    }
}
```

Demonstration

To demonstrate that the modified program can handle binary files:

1. Create a binary test file:

```
dd if=/dev/urandom of=binary_test.bin bs=1024 count=1
```

2. Run the modified `reverselines` program:

```
echo "binary_test.bin" | ./reverselines
```

3. Compare the original and reversed files:

```
cmp -l binary_test.bin binary_test.bin.rev
```

The `cmp` command show that the files differ only in the order of their lines, with newline characters inserted between them in the reversed file.(Result is put in the next page)

Conclusion

The `reverselines` application can indeed work with binary files, not just ASCII files. The only ASCII-specific assumption is the use of newline characters to delimit lines. By modifying the program to use `fgetc()` for reading and `fwrite()` for writing, we ensure that all binary data, including null characters, is handled correctly.

This demonstration shows that the core functionality of reversing lines in a file is not dependent on the file being ASCII text. The program treats each sequence of bytes between newline characters (or file boundaries) as a "line", regardless of the content of those bytes.

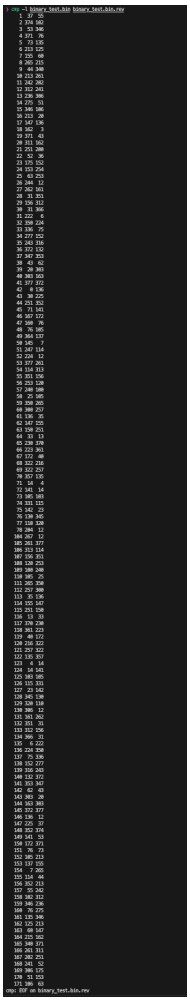

Figure 1: result