Here are some commonly used file handling system call codes:

- Open: To open a file.
  - System call: `int open(const char *pathname, int flags, mode_t mode);`
  - Code: `#include <fcntl.h>`
- Read: To read from a file.
  - System call: `ssize_t read(int fd, void *buf, size_t count);`
  - Code: `#include <unistd.h>`
- Write: To write to a file.
  - System call: `ssize_t write(int fd, const void *buf, size_t count);`
  - Code: `#include <unistd.h>`
- Close: To close a file.
  - System call: `int close(int fd);`
  - Code: `#include <unistd.h>`
- Create: To create a new file.
  - System call: `int creat(const char *pathname, mode_t mode);`
  - Code: `#include <fcntl.h>`
- Rename: To rename a file.
  - System call: `int rename(const char *oldpath, const char *newpath);`
  - Code: `#include <stdio.h>`
- Delete: To delete a file.
  - System call: `int unlink(const char *pathname);`
  - Code: `#include <unistd.h>`
- Seek: To move the file pointer within a file.
  - System call: `off_t lseek(int fd, off_t offset, int whence);`
  - Code: `#include <unistd.h>`
- Directory Manipulation: To work with directories.
  - System calls: `opendir`, `readdir`, `closedir`, `mkdir`, `rmdir`, etc.
  - Code: `#include <dirent.h>`
- File Information: To obtain information about a file.
  - System call: `int stat(const char *pathname, struct stat *buf);`
  - Code: `#include <sys/stat.h>`
- File Permissions: To set file permissions.
  - System call: `int chmod(const char *pathname, mode_t mode);`
  - Code: `#include <sys/stat.h>`

Q4.

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

int main() {
    // Create a new file (or overwrite if it exists)
    int fileDescriptor = open("example.txt", O_CREAT | O_WRONLY, 0644);

    if (fileDescriptor == -1) {
        perror("Error opening file");
        return 1;
    }

    // Write to the file
    const char *message = "Hello, world!\n";
    ssize_t bytesWritten = write(fileDescriptor, message, strlen(message));

    if (bytesWritten == -1) {
        perror("Error writing to file");
        close(fileDescriptor);
        return 1;
    }

    printf("Bytes written: %zd\n", bytesWritten);

    // Close the file
    if (close(fileDescriptor) == -1) {
        perror("Error closing file");
        return 1;
    }

    // Reopen the file for reading
    fileDescriptor = open("example.txt", O_RDONLY);

    if (fileDescriptor == -1) {
        perror("Error opening file for reading");
        return 1;
    }

    // Read from the file
    char buffer[100];
    ssize_t bytesRead = read(fileDescriptor, buffer, sizeof(buffer));

    if (bytesRead == -1) {
        perror("Error reading from file");
        close(fileDescriptor);
        return 1;
    }

    printf("Bytes read: %zd\n", bytesRead);
    printf("Content: %.*s", (int)bytesRead, buffer);

    // Close the file again
    if (close(fileDescriptor) == -1) {
        perror("Error closing file");
        return 1;
    }

    return 0;
}
```

Q3.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t child_pid;

    // Create a child process
    child_pid = fork();

    if (child_pid < 0) {
        perror("fork");
        exit(EXIT_FAILURE);
    } else if (child_pid == 0) {
        // This is the child process
        printf("Child process: My PID is %d\n", getpid());
        printf("Child process: My parent's PID is %d\n", getppid());
        sleep(3); // Simulating some work in the child process
        exit(EXIT_SUCCESS);
    } else {
        // This is the parent process
        printf("Parent process: My PID is %d\n", getpid());
        printf("Parent process: My child's PID is %d\n", child_pid);

        int status;
        pid_t terminated_child_pid = wait(&status);

        if (terminated_child_pid == -1) {
            perror("wait");
            exit(EXIT_FAILURE);
        }

        if (WIFEXITED(status)) {
            printf("Parent process: Child process with PID %d terminated normally with exit status %d\n",
                   terminated_child_pid, WEXITSTATUS(status));
        } else if (WIFSIGNALED(status)) {
            printf("Parent process: Child process with PID %d terminated by a signal: %d\n",
                   terminated_child_pid, WTERMSIG(status));
        }
    }

    return 0;
}
```

Q1

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

int main() {
    pid_t child_pid;

    // Create a child process
    child_pid = fork();

    if (child_pid < 0) {
        perror("fork");
        exit(EXIT_FAILURE);
    } else if (child_pid == 0) {
        // This is the child process
        printf("Child process: My PID is %d\n", getpid());
        printf("Child process: My parent's PID is %d\n", getppid());
        exit(EXIT_SUCCESS);
    } else {
        // This is the parent process
        printf("Parent process: My PID is %d\n", getpid());
        printf("Parent process: My child's PID is %d\n", child_pid);
    }

    return 0;
}
```

Q2

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

int main() {
    pid_t child_pid;

    child_pid = fork();

    if (child_pid == -1) {
        perror("fork");
        return 1;
    }

    if (child_pid == 0) { // Child process
        printf("Child process is executing.\n");
        printf("Child process will terminate now.\n");
        exit(0); // Terminate the child process
    } else { // Parent process
        printf("Parent process is waiting for the child process to finish.\n");
        wait(NULL); // Wait for the child process to finish
        printf("Parent process has completed.\n");
    }

    return 0;
}
```