```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <errno.h>

int main() {
    // ================================
    // 1. FILE OPERATIONS using open(), read(), write(), lseek(), close()
    // ================================

    // open() - Create/open a file
    int fd = open("example.txt", O_RDWR | O_CREAT | O_TRUNC, 0644);
    if (fd == -1) {
        perror("open failed");
        exit(1);
    }
    printf("File opened successfully. File descriptor: %d\n", fd);

    // write() - Write to file
    char *message = "Hello from system calls!\n";
    ssize_t bytes_written = write(fd, message, strlen(message));
    if (bytes_written == -1) {
        perror("write failed");
        close(fd);
        exit(1);
    }
    printf("Wrote %ld bytes to file\n", bytes_written);

    // lseek() - Move file pointer
    off_t offset = lseek(fd, 0, SEEK_SET);  // Go to beginning
    if (offset == -1) {
        perror("lseek failed");
        close(fd);
        exit(1);
    }
    printf("File pointer repositioned to: %ld\n", offset);

    // read() - Read from file
    char buffer[100];
```

```
ssize_t bytes_read = read(fd, buffer, sizeof(buffer) - 1);
if (bytes_read == -1) {
    perror("read failed");
    close(fd);
    exit(1);
}
buffer[bytes_read] = '\0';  // Null-terminate
printf("Read %ld bytes: %s", bytes_read, buffer);

// close() - Close file
if (close(fd) == -1) {
    perror("close failed");
    exit(1);
}
printf("File closed successfully\n\n");

// ================================
// 2. PROCESS CREATION using fork(), wait()
// ================================

printf("=== PROCESS DEMONSTRATION ===\n");

// fork() - Create child process
pid_t pid = fork();

if (pid == -1) {
    perror("fork failed");
    exit(1);
}

if (pid == 0) {
    // Child process
    printf("Child Process: PID = %d, Parent PID = %d\n",
           getpid(), getppid());

    // Child creates its own child (grandchild)
    pid_t grandchild_pid = fork();
    if (grandchild_pid == 0) {
        // Grandchild process
        printf("Grandchild Process: PID = %d, Parent PID = %d\n",
               getpid(), getppid());
        exit(0);  // Grandchild exits
    } else if (grandchild_pid > 0) {
        // Child waits for grandchild
```

```
            wait(NULL);
            printf("Child: Grandchild has finished\n");
        }
        exit(0);  // Child exits
    } else {
        // Parent process
        printf("Parent Process: PID = %d, Created Child PID = %d\n",
               getpid(), pid);

        // wait() - Parent waits for child to finish
        int status;
        wait(&status);
        printf("Parent: Child process finished with status: %d\n\n",
               WEXITSTATUS(status));
    }

    // ================================
    // 3. exec() DEMONSTRATION
    // ================================

    printf("=== EXEC DEMONSTRATION ===\n");

    pid_t exec_pid = fork();

    if (exec_pid == 0) {
        // Child process will execute ls command
        printf("Child about to execute 'ls -l'...\n");

        // exec() family - replace process image
        char *args[] = {"/bin/ls", "-l", NULL};
        execv(args[0], args);

        // If exec returns, it failed
        perror("exec failed");
        exit(1);
    } else {
        // Parent waits for exec child
        wait(NULL);
        printf("Exec demonstration complete\n\n");
    }

    // ================================
    // 4. FILE APPEND MODE demonstration
    // ================================
    printf("=== FILE APPEND DEMONSTRATION ===\n");

    // open() with O_APPEND
    int append_fd = open("append.txt", O_WRONLY | O_CREAT | O_APPEND, 0644);
    if (append_fd == -1) {
        perror("open append failed");
        exit(1);
    }

    // Write multiple times - all will append
    write(append_fd, "First line\n", 11);
    write(append_fd, "Second line\n", 12);
    write(append_fd, "Third line\n", 11);

    close(append_fd);
    printf("Three lines appended to append.txt\n\n");

    // ================================
    // 5. lseek() WITH DIFFERENT ORIGINS
    // ================================

    printf("=== LSEEK DEMONSTRATION ===\n");

    int lseek_fd = open("lseek_demo.txt", O_RDWR | O_CREAT | O_TRUNC, 0644);
    if (lseek_fd == -1) {
        perror("open lseek demo failed");
        exit(1);
    }

    // Write some text
    write(lseek_fd, "0123456789ABCDEFGHIJ", 20);

    // Demonstrate different lseek origins
    printf("File content: 0123456789ABCDEFGHIJ\n");

    // SEEK_SET: From beginning
    off_t pos1 = lseek(lseek_fd, 5, SEEK_SET);
    printf("After lseek(fd, 5, SEEK_SET): position = %ld (points to '5')\n", pos1);

    // SEEK_CUR: From current position
    off_t pos2 = lseek(lseek_fd, 3, SEEK_CUR);
    printf("After lseek(fd, 3, SEEK_CUR): position = %ld (points to '8')\n", pos2);
```

```
// SEEK_END: From end
off_t pos3 = lseek(lseek_fd, -4, SEEK_END);
printf("After lseek(fd, -4, SEEK_END): position = %ld (points to 'G')\n", pos3);

close(lseek_fd);

// ================================
// 6. ZOMBIE PROCESS PREVENTION with wait()
// ================================

printf("\n=== ZOMBIE PREVENTION DEMO ===\n");

pid_t zombie_pid = fork();

if (zombie_pid == 0) {
    // Child exits immediately
    printf("Child %d exiting...\n", getpid());
    exit(42);
} else {
    // Parent waits to prevent zombie
    sleep(1);  // Give child time to exit
    printf("Parent checking child status...\n");

    int child_status;
    pid_t terminated_pid = wait(&child_status);

    if (terminated_pid > 0) {
        printf("Child %d terminated. Exit status: %d\n",
               terminated_pid, WEXITSTATUS(child_status));
        printf("No zombie process created!\n");
    }
}

// ================================
// 7. chdir() EXAMPLE
// ================================

printf("\n=== CHDIR DEMONSTRATION ===\n");

char cwd[1024];
getcwd(cwd, sizeof(cwd));
printf("Current directory: %s\n", cwd);

// Try to change directory
```

```
    if (chdir("..") == 0) {
        getcwd(cwd, sizeof(cwd));
        printf("Changed to parent directory: %s\n", cwd);

        // Change back
        chdir(".");
    } else {
        perror("chdir failed");
    }

    printf("\n=== ALL SYSTEM CALLS DEMONSTRATED SUCCESSFULLY ===\n");

    // Clean up created files
    unlink("example.txt");
    unlink("append.txt");
    unlink("lseek_demo.txt");

    return 0;
}

# Basic compilation
gcc complete_system_calls.c -o system_calls_demo

# Simply execute
./system_calls_demo

# If permission denied, make it executable first
chmod +x system_calls_demo
./system_calls_demo

//=========================================================================

/************************************************************
 * COMPLETE PTHREAD PROGRAMMING REFERENCE
 * Includes: Threads, Mutex, Semaphore, Condition Variables
 * Compile: gcc -pthread -o threads threads.c
 ************************************************************/

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

```c
/* ============ 1. BASIC THREAD SYNTAX ============ */

/*
 * BASIC THREAD FUNCTION PROTOTYPE:
 * void *thread_function(void *arg);
 *
 * Key points:
 * 1. Must return void*
 * 2. Must accept void* argument
 * 3. Use pthread_exit() to return value
 */

// Example 1: Simple thread function
void *simple_thread(void *arg) {
    int thread_id = *((int *)arg);
    printf("Thread %d started\n", thread_id);

    for (int i = 0; i < 3; i++) {
        printf("Thread %d: Count %d\n", thread_id, i);
        sleep(1);
    }

    printf("Thread %d finished\n", thread_id);
    pthread_exit(NULL);  // Equivalent to return NULL;
}

// Example 2: Thread with return value
void *thread_with_return(void *arg) {
    int value = *((int *)arg);
    int *result = malloc(sizeof(int));
    *result = value * value;

    // Two ways to return value:
    // Method 1: pthread_exit
    // pthread_exit((void *)result);

    // Method 2: return statement
    return (void *)result;
}

// Example 3: Thread modifying shared data
int shared_counter = 0;
void *thread_modify_shared(void *arg) {
    int id = *((int *)arg);
    for (int i = 0; i < 10000; i++) {
        shared_counter++;  // UNSAFE: Race condition!
    }

    printf("Thread %d done. Shared counter: %d\n", id, shared_counter);
    return NULL;
}

/* ============ 2. MUTEX SYNTAX ============ */

// Global mutex declaration
pthread_mutex_t mutex_counter = PTHREAD_MUTEX_INITIALIZER;
int safe_counter = 0;

// Thread with mutex protection
void *thread_with_mutex(void *arg) {
    int id = *((int *)arg);

    for (int i = 0; i < 10000; i++) {
        // Method 1: Basic lock/unlock
        pthread_mutex_lock(&mutex_counter);
        safe_counter++;
        pthread_mutex_unlock(&mutex_counter);

        /* Alternative: Trylock (non-blocking)
        if (pthread_mutex_trylock(&mutex_counter) == 0) {
            safe_counter++;
            pthread_mutex_unlock(&mutex_counter);
        }
        */
    }

    printf("Thread %d done. Safe counter: %d\n", id, safe_counter);
    return NULL;
}

// Recursive mutex example (same thread can lock multiple times)
pthread_mutex_t recursive_mutex;
void recursive_function(int depth, int id) {
    if (depth <= 0) return;

    pthread_mutex_lock(&recursive_mutex);
    printf("Thread %d: Lock depth %d\n", id, depth);
    recursive_function(depth - 1, id);

    pthread_mutex_unlock(&recursive_mutex);
}

void *thread_recursive_mutex(void *arg) {
    int id = *((int *)arg);
    recursive_function(3, id);
    return NULL;
}

/* ============ 3. SEMAPHORE SYNTAX ============ */

#include <semaphore.h>

// Binary semaphore (like mutex but can be unlocked by different thread)
sem_t binary_sem;
int sem_counter = 0;

void *thread_with_semaphore(void *arg) {
    int id = *((int *)arg);

    for (int i = 0; i < 10000; i++) {
        sem_wait(&binary_sem);  // P operation (decrement)
        sem_counter++;
        sem_post(&binary_sem);  // V operation (increment)
    }

    printf("Thread %d done. Sem counter: %d\n", id, sem_counter);
    return NULL;
}

// Counting semaphore example (limits concurrent access)
sem_t counting_sem;
#define MAX_CONCURRENT 2

void *thread_limited_access(void *arg) {
    int id = *((int *)arg);

    sem_wait(&counting_sem);  // Wait for available slot
    printf("Thread %d: Entering critical section\n", id);
    sleep(2);
    printf("Thread %d: Leaving critical section\n", id);
    sem_post(&counting_sem);  // Release slot

    return NULL;
}

/* ============ 4. CONDITION VARIABLES ============ */

pthread_mutex_t cond_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t condition = PTHREAD_COND_INITIALIZER;
int ready = 0;

// Producer thread
void *producer_thread(void *arg) {
    sleep(1);  // Simulate work

    pthread_mutex_lock(&cond_mutex);
    ready = 1;
    printf("Producer: Data is ready!\n");

    // Signal one waiting thread
    pthread_cond_signal(&condition);

    // OR broadcast to all waiting threads
    // pthread_cond_broadcast(&condition);

    pthread_mutex_unlock(&cond_mutex);
    return NULL;
}

// Consumer thread
void *consumer_thread(void *arg) {
    int id = *((int *)arg);

    pthread_mutex_lock(&cond_mutex);

    while (ready == 0) {
        printf("Consumer %d: Waiting for data...\n", id);
        pthread_cond_wait(&condition, &cond_mutex);
    }

    printf("Consumer %d: Got the data!\n", id);
    pthread_mutex_unlock(&cond_mutex);

    return NULL;
```

```
}

/* ============ 5. THREAD ATTRIBUTES ============ */

void *thread_with_attr(void *arg) {
    int id = *((int *)arg);

    // Get thread attributes
    pthread_attr_t attr;
    size_t stack_size;
    pthread_attr_init(&attr);
    pthread_attr_getstacksize(&attr, &stack_size);

    printf("Thread %d: Stack size = %zu bytes\n", id, stack_size);

    pthread_attr_destroy(&attr);
    return NULL;
}

/* ============ 6. THREAD LOCAL STORAGE ============ */

// Thread-specific data
pthread_key_t tls_key;

void destroy_tls_data(void *data) {
    printf("Cleaning up thread-specific data: %p\n", data);
    free(data);
}

void *thread_with_tls(void *arg) {
    int id = *((int *)arg);

    // Allocate thread-specific data
    int *tls_data = malloc(sizeof(int));
    *tls_data = id * 100;

    // Associate with key
    pthread_setspecific(tls_key, tls_data);

    // Retrieve and use
    int *my_data = pthread_getspecific(tls_key);
    printf("Thread %d: TLS data = %d\n", id, *my_data);

    return NULL;
```

```
}

/* ============ 7. DETACHED THREADS ============ */

void *detached_thread(void *arg) {
    int id = *((int *)arg);
    printf("Detached thread %d running\n", id);
    sleep(2);
    printf("Detached thread %d exiting\n", id);
    return NULL;
}

/* ============ 8. COMPLETE EXAMPLE WITH ALL FEATURES ============ */

#define NUM_THREADS 5

// Thread arguments structure
typedef struct {
    int id;
    int iterations;
    pthread_mutex_t *mutex;
    sem_t *semaphore;
    int *shared_data;
} thread_args_t;

// Comprehensive thread function
void *comprehensive_thread(void *arg) {
    thread_args_t *args = (thread_args_t *)arg;

    printf("Thread %d starting\n", args->id);

    // Wait on semaphore
    sem_wait(args->semaphore);

    // Use mutex for critical section
    pthread_mutex_lock(args->mutex);
    printf("Thread %d in critical section\n", args->id);

    // Modify shared data
    for (int i = 0; i < args->iterations; i++) {
        (*args->shared_data)++;
    }

    pthread_mutex_unlock(args->mutex);
```

```c
    // Signal semaphore
    sem_post(args->semaphore);

    // Return thread-specific result
    int *result = malloc(sizeof(int));
    *result = args->id * 1000;

    printf("Thread %d finishing\n", args->id);
    return result;
}

/* ============ MAIN FUNCTION - DEMO ALL EXAMPLES ============ */

int main() {
    printf("\n=== C PTHREAD PROGRAMMING REFERENCE ===\n\n");

    /* ===== 1. BASIC THREAD CREATION ===== */
    printf("1. BASIC THREAD CREATION:\n");

    pthread_t basic_threads[2];
    int thread_ids[2] = {1, 2};

    // Create threads
    for (int i = 0; i < 2; i++) {
        pthread_create(&basic_threads[i], NULL, simple_thread, &thread_ids[i]);
    }

    // Join threads
    for (int i = 0; i < 2; i++) {
        pthread_join(basic_threads[i], NULL);
    }
    printf("\n");

    /* ===== 2. THREAD WITH RETURN VALUE ===== */
    printf("2. THREAD WITH RETURN VALUE:\n");

    pthread_t return_thread;
    int input_value = 5;
    void *return_result;

    pthread_create(&return_thread, NULL, thread_with_return, &input_value);
    pthread_join(return_thread, &return_result);
```

```
int *result = (int *)return_result;
printf("Thread returned: %d (5 * 5 = 25)\n", *result);
free(result);
printf("\n");

/* ===== 3. RACE CONDITION DEMO ===== */
printf("3. RACE CONDITION DEMO:\n");

pthread_t race_threads[2];
int race_ids[2] = {1, 2};
shared_counter = 0;

for (int i = 0; i < 2; i++) {
    pthread_create(&race_threads[i], NULL, thread_modify_shared, &race_ids[i]);
}

for (int i = 0; i < 2; i++) {
    pthread_join(race_threads[i], NULL);
}

printf("Final shared counter (unsafe): %d (Expected: 20000)\n", shared_counter);
printf("\n");

/* ===== 4. MUTEX PROTECTION ===== */
printf("4. MUTEX PROTECTION:\n");

pthread_t mutex_threads[2];
int mutex_ids[2] = {1, 2};
safe_counter = 0;

// Initialize mutex (alternative to PTHREAD_MUTEX_INITIALIZER)
pthread_mutex_init(&mutex_counter, NULL);

for (int i = 0; i < 2; i++) {
    pthread_create(&mutex_threads[i], NULL, thread_with_mutex, &mutex_ids[i]);
}

for (int i = 0; i < 2; i++) {
    pthread_join(mutex_threads[i], NULL);
}

printf("Final safe counter: %d (Expected: 20000)\n", safe_counter);

// Clean up mutex
```

```
    pthread_mutex_destroy(&mutex_counter);
    printf("\n");

    /* ===== 5. SEMAPHORE DEMO ===== */
    printf("5. SEMAPHORE DEMO:\n");

    pthread_t sem_threads[2];
    int sem_ids[2] = {1, 2};
    sem_counter = 0;

    // Initialize binary semaphore
    sem_init(&binary_sem, 0, 1);  // 0 = not shared between processes, 1 = initial value

    for (int i = 0; i < 2; i++) {
        pthread_create(&sem_threads[i], NULL, thread_with_semaphore, &sem_ids[i]);
    }

    for (int i = 0; i < 2; i++) {
        pthread_join(sem_threads[i], NULL);
    }

    printf("Final semaphore counter: %d (Expected: 20000)\n", sem_counter);

    // Clean up semaphore
    sem_destroy(&binary_sem);
    printf("\n");

    /* ===== 6. COUNTING SEMAPHORE ===== */
    printf("6. COUNTING SEMAPHORE (Limited to %d concurrent threads):\n",
MAX_CONCURRENT);

    pthread_t limited_threads[4];
    int limited_ids[4] = {1, 2, 3, 4};

    sem_init(&counting_sem, 0, MAX_CONCURRENT);

    for (int i = 0; i < 4; i++) {
        pthread_create(&limited_threads[i], NULL, thread_limited_access, &limited_ids[i]);
    }

    for (int i = 0; i < 4; i++) {
        pthread_join(limited_threads[i], NULL);
    }
```

```
sem_destroy(&counting_sem);
printf("\n");

/* ===== 7. CONDITION VARIABLES ===== */
printf("7. CONDITION VARIABLES:\n");

pthread_t producer, consumer1, consumer2;
int consumer_ids[2] = {1, 2};

ready = 0;
pthread_mutex_init(&cond_mutex, NULL);
pthread_cond_init(&condition, NULL);

pthread_create(&consumer1, NULL, consumer_thread, &consumer_ids[0]);
pthread_create(&consumer2, NULL, consumer_thread, &consumer_ids[1]);
pthread_create(&producer, NULL, producer_thread, NULL);

pthread_join(producer, NULL);
pthread_join(consumer1, NULL);
pthread_join(consumer2, NULL);

pthread_mutex_destroy(&cond_mutex);
pthread_cond_destroy(&condition);
printf("\n");

/* ===== 8. THREAD LOCAL STORAGE ===== */
printf("8. THREAD LOCAL STORAGE:\n");

pthread_t tls_threads[3];
int tls_ids[3] = {1, 2, 3};

// Create TLS key
pthread_key_create(&tls_key, destroy_tls_data);

for (int i = 0; i < 3; i++) {
   pthread_create(&tls_threads[i], NULL, thread_with_tls, &tls_ids[i]);
}

for (int i = 0; i < 3; i++) {
   pthread_join(tls_threads[i], NULL);
}

// Clean up TLS key
pthread_key_delete(tls_key);
```

```
printf("\n");

/* ===== 9. DETACHED THREADS ===== */
printf("9. DETACHED THREADS:\n");

pthread_t detached;
int detach_id = 99;
pthread_attr_t attr;

pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

pthread_create(&detached, &attr, detached_thread, &detach_id);

// No need to join detached threads
pthread_attr_destroy(&attr);

// Give detached thread time to finish
sleep(3);
printf("Main thread continues after detached thread\n\n");

/* ===== 10. COMPREHENSIVE EXAMPLE ===== */
printf("10. COMPREHENSIVE EXAMPLE (All features):\n");

pthread_t comp_threads[NUM_THREADS];
thread_args_t thread_args[NUM_THREADS];
int shared_data = 0;

// Initialize synchronization primitives
pthread_mutex_t comp_mutex = PTHREAD_MUTEX_INITIALIZER;
sem_t comp_sem;
sem_init(&comp_sem, 0, 2);  // Allow 2 threads in critical section

// Create threads with comprehensive arguments
for (int i = 0; i < NUM_THREADS; i++) {
    thread_args[i].id = i + 1;
    thread_args[i].iterations = 1000;
    thread_args[i].mutex = &comp_mutex;
    thread_args[i].semaphore = &comp_sem;
    thread_args[i].shared_data = &shared_data;

    pthread_create(&comp_threads[i], NULL, comprehensive_thread, &thread_args[i]);
}
```

```
    // Collect results
    void *thread_returns[NUM_THREADS];
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(comp_threads[i], &thread_returns[i]);

        int *ret = (int *)thread_returns[i];
        printf("Thread %d returned: %d\n", i + 1, *ret);
        free(ret);
    }

    printf("Final shared data value: %d (Expected: %d)\n",
           shared_data, NUM_THREADS * 1000);

    // Clean up
    pthread_mutex_destroy(&comp_mutex);
    sem_destroy(&comp_sem);

    printf("\n=== ALL EXAMPLES COMPLETED SUCCESSFULLY ===\n");

    return 0;
}

/****************************************************************
 * QUICK REFERENCE - COMMON PTHREAD PATTERNS
 ****************************************************************

=== 1. THREAD CREATION PATTERNS ===

// Pattern 1: Simple thread
pthread_t thread;
int thread_id = 1;
pthread_create(&thread, NULL, thread_func, &thread_id);
pthread_join(thread, NULL);

// Pattern 2: Multiple threads
pthread_t threads[N];
int ids[N];
for (int i = 0; i < N; i++) {
    ids[i] = i;
    pthread_create(&threads[i], NULL, thread_func, &ids[i]);
}
for (int i = 0; i < N; i++) {
    pthread_join(threads[i], NULL);
}
```

```
=== 2. SYNCHRONIZATION PATTERNS ===

// Pattern 1: Mutex for critical section
pthread_mutex_lock(&mutex);
// Critical section
pthread_mutex_unlock(&mutex);

// Pattern 2: Binary semaphore
sem_wait(&sem);  // Entry
// Critical section
sem_post(&sem);  // Exit

// Pattern 3: Condition variable
pthread_mutex_lock(&mutex);
while (!condition) {
    pthread_cond_wait(&cond, &mutex);
}
// Condition is true
pthread_mutex_unlock(&mutex);

=== 3. THREAD COMMUNICATION PATTERNS ===

// Pattern 1: Return value
void *result;
pthread_join(thread, &result);
int value = *(int *)result;
free(result);

// Pattern 2: Thread-local storage
pthread_setspecific(key, data);
void *data = pthread_getspecific(key);

=== 4. ERROR HANDLING PATTERNS ===

int rc = pthread_create(&thread, NULL, func, arg);
if (rc) {
    fprintf(stderr, "Error creating thread: %d\n", rc);
    exit(EXIT_FAILURE);
}

**************************************************************
 * COMMON MISTAKES TO AVOID
 **************************************************************
```

```
1. NOT JOINING THREADS: Causes memory leaks
2. PASSING LOCAL VARIABLES: Variables must outlive thread
3. MISSING MUTEX UNLOCK: Causes deadlock
4. BUSY WAITING: Use condition variables instead
5. NOT CHECKING RETURN VALUES: Always check pthread_* returns

******************************************************************
 * COMPILATION & EXECUTION
 ******************************************************************

// Compile:
// gcc -pthread -o threads threads.c

// Run:
// ./threads

// Debug with Valgrind:
// valgrind --tool=helgrind ./threads
*/
```