1. Create a multithreaded C program that sorts an array of integers using two threads. The first thread should sort the first half of the array, and the second thread should sort the second half. After both threads finish sorting, the main thread should merge the two sorted halves into a final sorted array.

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
#define SIZE 10
int arr[SIZE];
// Bubble sort
void bubble_sort(int *start, int size) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            if (start[j] > start[j + 1]) {
                int temp = start[j];
                start[j] = start[j + 1];
                start[j + 1] = temp;
            }
        }
    }
}
```

```c
void merge(int *arr, int *left, int left_size, int *right, int right_size) {
    int i = 0, j = 0, k = 0;
    while (i < left_size && j < right_size) {
        if (left[i] <= right[j]) {
            arr[k++] = left[i++];
        } else {
            arr[k++] = right[j++];
        }
    }
    while (i < left_size) {
        arr[k++] = left[i++];
    }
    while (j < right_size) {
        arr[k++] = right[j++];
    }
}
```

```c
void *sort_first_half(void *arg) {
    int half_size = SIZE / 2;
    bubble_sort(arr, half_size);
    return NULL;
}
void *sort_second_half(void *arg) {
    int half_size = SIZE / 2;
    bubble_sort(arr + half_size, SIZE - half_size); // Handles odd SIZE
    return NULL;
}
int main() {
    srand(time(NULL));
    for (int i = 0; i < SIZE; i++) {
        arr[i] = rand() % 100;
    }
    printf("Unsorted array:\n");
    for (int i = 0; i < SIZE; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
```

```c
    pthread_t thread1, thread2;
    pthread_create(&thread1, NULL, sort_first_half, NULL);
    pthread_create(&thread2, NULL, sort_second_half, NULL);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    int *left = arr;
    int *right = arr + SIZE / 2;
    int left_size = SIZE / 2;
    int right_size = SIZE - left_size;

    int *merged = (int *)malloc(sizeof(int) * SIZE);
    merge(merged, left, left_size, right, right_size);
    printf("Merged sorted array:\n");
    for (int i = 0; i < SIZE; i++) {
        printf("%d ", merged[i]);
    }
    printf("\n");
    free(merged);
    return 0;
}
```

2. Create a multithreaded C program that performs matrix multiplication using threads. Prompt the user to input two square matrices of size 3x3. Use a separate thread to calculate each row of the result matrix. After all threads have completed, display the final multiplied matrix.

```c
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<unistd.h>
#define SIZE 3
#define NUM_THREADS 3
int matrixA[SIZE][SIZE];
int matrixB[SIZE][SIZE];
int result[SIZE][SIZE];

void *funct(void *arg) {
    int row = *(int *)arg;
    for(int col=0;col<SIZE;col++) {
        result[row][col] = 0;
        for(int k=0;k<SIZE;k++) {
            result[row][col] += matrixA[row][k] * matrixB[k][col];
        }
    }
    return NULL;
}
```

```c
int main() {
    pthread_t threadids[NUM_THREADS];
    int thread_args[NUM_THREADS];
    // Initialize matrices
    printf("Enter the elements of matrix A:\n");
    for(int i=0;i<SIZE;i++) {
        for(int j=0;j<SIZE;j++) {
            scanf("%d", &matrixA[i][j]);
        }
    }
    printf("Enter the elements of matrix B:\n");
    for(int i=0;i<SIZE;i++) {
        for(int j=0;j<SIZE;j++) {
         scanf("%d", &matrixB[i][j]);
        }
    }
```

```c
for(int i=0;i<NUM_THREADS;i++) {
    thread_args[i] = i;
    if(pthread_create(&threadids[i],NULL,funct,(void *) &thread_args[i]) !=0) {
        perror("Error creating thread");
        exit(1);
    }
}
```

```c
    for(int i=0;i<NUM_THREADS;i++) {
        pthread_join(threadids[i],NULL);
    }
    printf("Resultant matrix:\n");
    for(int i=0;i<SIZE;i++) {
        for(int j=0;j<SIZE;j++) {
            printf("%d ", result[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

4. Create a C program that continuously prints a message like “Running…” every 2 seconds. When the user presses Ctrl+C (SIGINT), catch the signal and display “SIGINT caught, terminating safely.” Use `signal()` to handle the signal.

```c
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<unistd.h>
#include<signal.h>

void signal_handler(int signum) {
    printf("Caught signal %d\n", signum);
    exit(0);
}
int main() {
    signal(SIGINT,signal_handler);
    while(1) {
        printf("Running...\n");
        sleep(2);

    }
    return 0;
}
```

5. Develop a C program that simulates a countdown timer from 10. If the user presses Ctrl+C, catch SIGINT and pause the timer. When the user presses Ctrl+Z, catch SIGTSTP to resume the countdown.

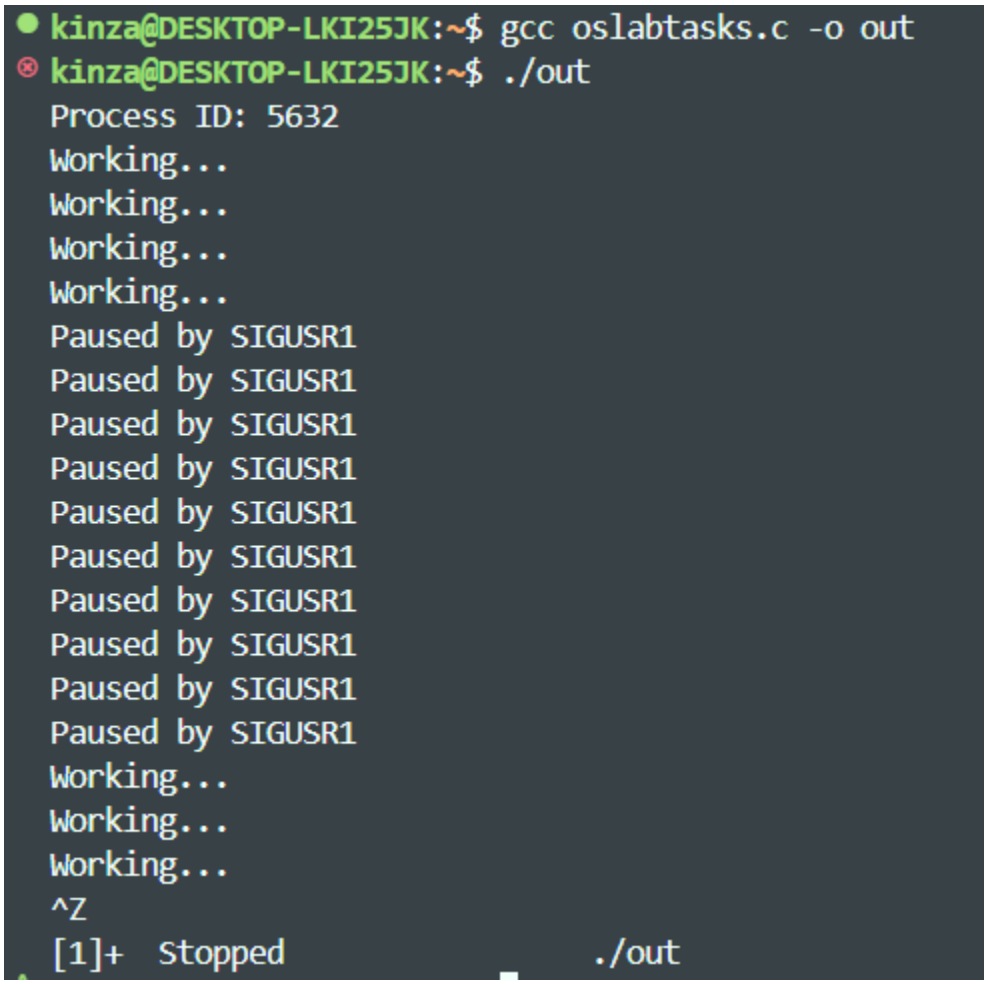
```c
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<unistd.h>
#include<signal.h>
#include<stdbool.h>
bool paused=1;
void signal_stsp(int signum) {
    paused =0;
    printf("Caught signal %d. Press Ctrl+C to pause. \n", signum);
}
void signal_handler(int signum) {
    paused=1;
    printf("Caught signal %d. Press Ctrl+Z to resume. \n", signum);
}
int main() {
    signal(SIGINT,signal_handler);
    signal(SIGTSTP,signal_stsp);
    int i=10;
    while(i>0) {
        if(!paused) {
            printf("Running...\n");
            sleep(1);
            i--;
        } else {
            printf("Paused...\n");
            sleep(1);
        }
    }
    return 0;
}
```

6. Create a C program that runs an infinite loop, printing “Working...” every 3 seconds. Set up a signal handler for SIGUSR1 such that when the signal is received, a global flag is toggled between 1 and 0. If the flag is 1, the process should print “Paused by SIGUSR1” instead of “Working...”. Sending the signal again should resume the normal output.

```c
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<unistd.h>
#include<signal.h>
#include<stdbool.h>
volatile sig_atomic_t flag= 0;
void signal_handler(int signum) {
    if (signum == SIGUSR1) {
        flag = !flag; // Toggle the flag
    }
}
int main() {
    signal(SIGUSR1,signal_handler);
    printf("Process ID: %d\n", getpid());
    while(1) {
        if(!flag){
        printf("Working...\n");
        sleep(3);
        } else {
            printf("Paused by SIGUSR1\n");
            sleep(3);
        }
    }
    return 0;
}
```

```
kinza@DESKTOP-LKI25JK:~$ kill -SIGUSR1 5632
kinza@DESKTOP-LKI25JK:~$ kill -SIGUSR1 5632
kinza@DESKTOP-LKI25JK:~$
```

Sending the signal kill -SIGUSR1 <pid> is used to pause and sending the signal kill -SIGUSR1 <pid> resumes it.



7. Create a C program that uses semaphores to control access to a shared counter variable updated by multiple threads. Ensure that only one thread can update the counter at a time.

```c
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<unistd.h>
#include<semaphore.h>
#define NUM_THREADS 5
int counter =0;
sem_t sem;

void *func(void *arg) {
    int i = *(int *) arg;
    sem_wait(&sem);
    //Critical section
    printf("Thread %d: Counter before increment: %d\n", i, counter);
    counter++;
    printf("Thread %d: Counter after increment: %d\n", i, counter);
    sem_post(&sem);
    pthread_exit(NULL);
}
```

```c
int main(){
    sem_init(&sem,0,1);
    pthread_t threads[NUM_THREADS];
    int thread_args[NUM_THREADS];
    for(int i=0;i<NUM_THREADS;i++){
       thread_args[i] = i;
       if(pthread_create(&threads[i], NULL, func, (void *)&thread_args[i]) != 0) {
           perror("Failed to create thread");
           exit(EXIT_FAILURE);
       }
    }
    for(int i=0;i<NUM_THREADS;i++){
       pthread_join(threads[i], NULL);
    }
    printf("Final Counter Value: %d\n", counter);
    sem_destroy(&sem);
    printf("Semaphore destroyed\n");
    return 0;
}
```

```
kinza@DESKTOP-LKI25JK:~$ gcc oslabtasks.c -o out
kinza@DESKTOP-LKI25JK:~$ ./out
Thread 0: Counter before increment: 0
Thread 0: Counter after increment: 1
Thread 1: Counter before increment: 1
Thread 1: Counter after increment: 2
Thread 2: Counter before increment: 2
Thread 2: Counter after increment: 3
Thread 3: Counter before increment: 3
Thread 3: Counter after increment: 4
Thread 4: Counter before increment: 4
Thread 4: Counter after increment: 5
Final Counter Value: 5
Semaphore destroyed
```

9. Create a C program where multiple threads increment a global variable. Use a mutex to ensure the updates are synchronized and no race condition occurs.

```c
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<unistd.h>
#define NUM_THREADS 5
int counter =0;
pthread_mutex_t mutex;

void *func(void *arg) {
    int i = *(int *) arg;
    pthread_mutex_lock(&mutex);
    //Critical section
    printf("Thread %d: Counter before increment: %d\n", i, counter);
    counter++;
    printf("Thread %d: Counter after increment: %d\n", i, counter);
    pthread_mutex_unlock(&mutex);
    pthread_exit(NULL);
}
```

```c
int main(){
    pthread_mutex_init(&mutex,NULL);
    pthread_t threads[NUM_THREADS];
    int thread_args[NUM_THREADS];
    for(int i=0;i<NUM_THREADS;i++){
       thread_args[i] = i;
       if(pthread_create(&threads[i], NULL, func, (void *)&thread_args[i]) != 0) {
           perror("Failed to create thread");
           exit(EXIT_FAILURE);
       }
    }
    for(int i=0;i<NUM_THREADS;i++){
       pthread_join(threads[i], NULL);
    }
    printf("Final Counter Value: %d\n", counter);
    pthread_mutex_destroy(&mutex);
    printf("Mutex destroyed\n");
    return 0;
}
```

```
kinza@DESKTOP-LKI25JK:~$ gcc oslabtasks.c -o out
kinza@DESKTOP-LKI25JK:~$ ./out
Thread 1: Counter before increment: 0
Thread 1: Counter after increment: 1
Thread 3: Counter before increment: 1
Thread 3: Counter after increment: 2
Thread 4: Counter before increment: 2
Thread 4: Counter after increment: 3
Thread 0: Counter before increment: 3
Thread 0: Counter after increment: 4
Thread 2: Counter before increment: 4
Thread 2: Counter after increment: 5
Final Counter Value: 5
Mutex destroyed
```

10. Create a program where multiple threads simulate people accessing an ATM. Use a mutex to restrict access to one person at a time, printing messages when each user accesses and leaves the ATM.

```c
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<unistd.h>
#define NUM_THREADS 5
int balance =0;
pthread_mutex_t mutex;

void *func(void *arg) {
    int i = *(int *) arg;
    pthread_mutex_lock(&mutex);
    //Critical section
    printf("Person %d accessing the ATM \n",i);
    printf("Enter 1. for withdrawal and 2. for deposit\n");
    int choice;
    scanf("%d",&choice);
    switch(choice){
        case 1:
            printf("Enter amount to withdraw\n");
            int withdraw;
            scanf("%d",&withdraw);
            if(withdraw>balance){
                printf("Insufficient balance\n");
            }else{
                balance-=withdraw;
                printf("Withdrawal successful. New balance: %d\n",balance);
            }
            break;
        case 2:
            printf("Enter amount to deposit\n");
            int deposit;
            scanf("%d",&deposit);
            balance+=deposit;
            printf("Deposit successful. New balance: %d\n",balance);
            break;
        default:
            printf("Invalid choice\n");
    }
    pthread_mutex_unlock(&mutex);
    pthread_exit(NULL);
}
```

```c
int main(){
    pthread_mutex_init(&mutex,NULL);
    pthread_t threads[NUM_THREADS];
    int thread_args[NUM_THREADS];
    for(int i=0;i<NUM_THREADS;i++){
        thread_args[i] = i;
        if(pthread_create(&threads[i], NULL, func, (void *)&thread_args[i]) != 0) {
            perror("Failed to create thread");
            exit(EXIT_FAILURE);
        }
    }
    for(int i=0;i<NUM_THREADS;i++){
        pthread_join(threads[i], NULL);
    }
    printf("Final Balance Value: %d\n", balance);
    pthread_mutex_destroy(&mutex);
    printf("Mutex destroyed\n");
    return 0;
}
```

```
kinza@DESKTOP-LKI25JK:~$ gcc oslabtasks.c -o out
kinza@DESKTOP-LKI25JK:~$ ./out
Person 1 accessing the ATM
Enter 1. for withdrawal and 2. for deposit
2
Enter amount to deposit
120
Deposit successful. New balance: 120
Person 2 accessing the ATM
Enter 1. for withdrawal and 2. for deposit
1
Enter amount to withdraw
50
Withdrawal successful. New balance: 70
Person 0 accessing the ATM
Enter 1. for withdrawal and 2. for deposit
2
Enter amount to deposit
100
Deposit successful. New balance: 170
Person 3 accessing the ATM
Enter 1. for withdrawal and 2. for deposit
2
Enter amount to deposit
100
Deposit successful. New balance: 270
Person 4 accessing the ATM
Enter 1. for withdrawal and 2. for deposit
2
Enter amount to deposit
130
Deposit successful. New balance: 400
Final Balance Value: 400
Mutex destroyed
```

3. Write a C program that replicates a simplified version of the `cp` command using system calls. It should read the source file using `read()` and write to a destination file using `write()`, with error handling.

```c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<unistd.h>
#include<fcntl.h>
#define SIZE 1024
int main(int argc,char *argv[]) {
    if(argc!=3) {
        printf("Usage: %s <source_file> <destination_file>\n",argv[0]);
        exit(EXIT_FAILURE);
    }
    int srcfd = open(argv[1],O_RDONLY);
    if(srcfd<0) {
        perror("Error opening source file");
        exit(EXIT_FAILURE);
    }
    int destfd = open(argv[2],O_WRONLY|O_CREAT|O_TRUNC,0644);
    if(destfd<0) {
        perror("Error opening destination file");
        close(srcfd);
        exit(EXIT_FAILURE);
    }
    char buffer[SIZE];
    ssize_t bytesread;
```

```c
while((bytesread = read(srcfd,buffer,SIZE)) > 0) {
    ssize_t byteswritten = write(destfd,buffer,bytesread);
    if(byteswritten<0) {
        perror("Error writing to destination file");
        close(srcfd);
        close(destfd);
        exit(EXIT_FAILURE);
    }
    if(byteswritten<bytesread) {
        perror("Error: Not all bytes written to destination file");
        close(srcfd);
        close(destfd);
        exit(EXIT_FAILURE);
    }
}
```

```
    if(bytesread<0) {
        perror("Error reading from source file");
        close(srcfd);
        close(destfd);
        exit(EXIT_FAILURE);
    }
    close(srcfd);
    close(destfd);
    return 0;
}
```

```
kinza@DESKTOP-LKI25JK:~$ gcc oslabtasks.c -o out
kinza@DESKTOP-LKI25JK:~$ ./out synchro.c dest.c
```

8. Implement the Dining Philosophers problem using semaphores. Ensure no deadlock occurs and neighboring philosophers do not eat simultaneously.

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

#define N 5
#define THINKING 0
#define HUNGRY 1
#define EATING 2
#define LEFT(i) ((i + N - 1) % N)
#define RIGHT(i) ((i + 1) % N)

int state[N];
sem_t mutex;
sem_t S[N];
```

```
void test(int i) {
    if(state[i] == HUNGRY && state[LEFT(i)] != EATING && state[RIGHT(i)] != EATING) {
        state[i] = EATING;
        sleep(2);
        printf("Philosopher %d is eating\n", i + 1);
        sem_post(&S[i]);
    }
}
```

```
void takeforks(int i) {
    sem_wait(&mutex);
    state[i] = HUNGRY;
    printf("Philosopher %d is hungry\n",i+1);
    test(i);
    sem_post(&mutex);
    sem_wait(&S[i]);
}
```

```
void putforks(int i) {
    sem_wait(&mutex);
    state[i] = THINKING;
    printf("Philosopher %d is thinking\n",i+1);
    test(LEFT(i));
    test(RIGHT(i));
    sem_post(&mutex);
}
```

```
void *philosopher(void *num) {
    while(1) {
        int i = (*(int *)num);
        sleep(1);
        takeforks(i);
        sleep(0);
        putforks(i);
    }
}
```

```c
int main() {
    int i;
    pthread_t thread_id[N];
    int thread_args[N];
    sem_init(&mutex, 0, 1);
    for(i = 0; i < N; i++) {
        state[i] = THINKING;
        sem_init(&S[i], 0, 0);
    }
    for(i = 0; i < N; i++) {
        thread_args[i] = i;
        pthread_create(&thread_id[i], NULL, philosopher,(void *) &thread_args[i]);
        printf("Philosopher %d is thinking\n", i + 1);
    }
    for(i = 0; i < N; i++) {
        pthread_join(thread_id[i], NULL);
    }
}
```

```
kinza@DESKTOP-LKI25JK:~$ gcc oslabtasks.c -o out
kinza@DESKTOP-LKI25JK:~$ ./out
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
Philosopher 1 is hungry
Philosopher 1 is eating
Philosopher 3 is hungry
Philosopher 3 is eating
Philosopher 5 is hungry
Philosopher 2 is hungry
Philosopher 4 is hungry
Philosopher 1 is thinking
Philosopher 5 is eating
Philosopher 3 is thinking
Philosopher 2 is eating
Philosopher 5 is thinking
Philosopher 4 is eating
Philosopher 1 is hungry
Philosopher 2 is thinking
Philosopher 1 is eating
Philosopher 3 is hungry
Philosopher 1 is thinking
Philosopher 5 is hungry
Philosopher 4 is thinking
Philosopher 3 is eating
^Z
[3]+  Stopped                 ./out
```

4. Design a program where multiple customer threads place orders, and limited kitchen threads (2 chefs) prepare food. Use semaphores to limit the number of active chefs and mutex to handle shared order data.

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#define MAX_ORDERS 10
#define NUM_CUSTOMERS 5

typedef struct {
    int order_id;
    int customer_id;
} Order;
Order order_queue[MAX_ORDERS];
int order_count = 0;
pthread_mutex_t order_mutex;
sem_t chef_semaphore;

void* customer_thread(void* arg) {
    int customer_id = *((int*)arg);
    free(arg);
    pthread_mutex_lock(&order_mutex);
    if (order_count < MAX_ORDERS) {
        order_queue[order_count].order_id = order_count + 1;
        order_queue[order_count].customer_id = customer_id;
        printf("Customer %d placed Order %d\n", customer_id, order_count + 1);
        order_count++;
    } else {
        printf("Customer %d could not place order (Queue full)\n", customer_id);
    }
    pthread_mutex_unlock(&order_mutex);
    return NULL;
}
```

```c
void* chef_thread(void* arg) {
    while (1) {
        sem_wait(&chef_semaphore); // Only 2 chefs allowed here
        pthread_mutex_lock(&order_mutex);

        if (order_count > 0) {
            Order order = order_queue[--order_count];
            pthread_mutex_unlock(&order_mutex);
            printf("Chef is preparing Order %d from Customer %d...\n",
                order.order_id, order.customer_id);
            sleep(1);
            printf("Chef completed Order %d\n", order.order_id);
        } else {
            pthread_mutex_unlock(&order_mutex);
            sem_post(&chef_semaphore);
            break;
        }
        sem_post(&chef_semaphore);
    }
    return NULL;
}
```

```c
int main() {
    pthread_t customers[NUM_CUSTOMERS];
    pthread_t chefs[2];
    pthread_mutex_init(&order_mutex, NULL);
    sem_init(&chef_semaphore, 0, 2); // Only 2 chefs
    // Create customer threads
    for (int i = 0; i < NUM_CUSTOMERS; ++i) {
        int* id = malloc(sizeof(int));
        *id = i + 1;
        pthread_create(&customers[i], NULL, customer_thread, id);
    }
    // Wait for all customers to finish
    for (int i = 0; i < NUM_CUSTOMERS; ++i) {
        pthread_join(customers[i], NULL);
    }
```

```
    // Create chef threads
    for (int i = 0; i < 2; ++i) {
        pthread_create(&chefs[i], NULL, chef_thread, NULL);
    }
    // Wait for chefs to finish
    for (int i = 0; i < 2; ++i) {
        pthread_join(chefs[i], NULL);
    }
    pthread_mutex_destroy(&order_mutex);
    sem_destroy(&chef_semaphore);
    return 0;
}
```

```
kinza@DESKTOP-LKI25JK:~$ gcc synchro.c -o out -pthread
kinza@DESKTOP-LKI25JK:~$ ./out
Customer 1 placed Order 1
Customer 2 placed Order 2
Customer 4 placed Order 3
Customer 5 placed Order 4
Customer 3 placed Order 5
Chef is preparing Order 5 from Customer 3...
Chef is preparing Order 4 from Customer 5...
Chef completed Order 5
Chef is preparing Order 3 from Customer 4...
Chef completed Order 4
Chef is preparing Order 2 from Customer 2...
Chef completed Order 2
Chef is preparing Order 1 from Customer 1...
Chef completed Order 3
Chef completed Order 1
kinza@DESKTOP-LKI25JK:~$
```

1. Write a C program that creates a temporary file and continuously writes data to it every second. The program should handle the SIGINT signal (Ctrl+C). Upon receiving the signal, it should close the file, delete it from disk, and display a message confirming cleanup before termination.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <signal.h>
#include <string.h>
int fd;
void handle_sigint(int sig) {
    close(fd);

    remove("temp.txt");
    printf("Signal %d received. Closing file and deleting it.\n", sig);
    printf("\nFile closed and deleted. Exiting program.\n");
    exit(0);
}
int main() {
    signal(SIGINT, handle_sigint);
    fd = open("temp.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fd == -1) {
        perror("Error opening file");
        return 1;
    }
    int i = 1;
    char buffer[50];
    while (1) {
        int len = sprintf(buffer, "Line %d\n", i++);
        write(fd, buffer, len);
        sleep(1);
    }
    return 0;
}
```

temp.txt

```
Line 1
Line 2
Line 3
Line 4
Line 5
Line 6
Line 7
Line 8
Line 9
Line 10
Line 11
Line 12
Line 13
Line 14
Line 15
```

```
kinza@DESKTOP-LKI25JK:~$ gcc synchro.c -o out -pthread
kinza@DESKTOP-LKI25JK:~$ ./out
^CSignal 2 received. Closing file and deleting it.

File closed and deleted. Exiting program.
```

```
while (fgets(buffer, sizeof(buffer), stdin)) {
    readbytes = strlen(buffer);
    write(fd, buffer, readbytes);
}
```

BOUNDED BUFFER PROBLEM

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include<unistd.h>
#define BUFFER_SIZE 5
sem_t mutex, empty, full;
int buffer[BUFFER_SIZE];
int in = 0, out = 0;
void *producer(void *arg) {
int item;
while (1) {
item = rand() % 100;
sem_wait(&empty);
sem_wait(&mutex);
buffer[in] = item;
printf("Produced: %d\n", item);
in = (in + 1) % BUFFER_SIZE;
sem_post(&mutex);
sem_post(&full);
sleep(rand() % 3);
}
}
```

```c
void *consumer(void *arg) {
int item;
while (1) {
sem_wait(&full);
sem_wait(&mutex);
item = buffer[out];
printf("Consumed: %d\n", item);
out = (out + 1) % BUFFER_SIZE;
sem_post(&mutex);
sem_post(&empty);
sleep(rand() % 3);
}
}
```

```c
int main() {
pthread_t producer_thread, consumer_thread;

sem_init(&mutex, 0, 1);
sem_init(&empty, 0, BUFFER_SIZE);
sem_init(&full, 0, 0);
pthread_create(&producer_thread, NULL, producer, NULL);
pthread_create(&consumer_thread, NULL, consumer, NULL);
pthread_join(producer_thread, NULL);
pthread_join(consumer_thread, NULL);
sem_destroy(&mutex);
sem_destroy(&empty);
sem_destroy(&full);
return 0;
}
```

READERS WRITERS PROBLEM

```c
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<semaphore.h>
#include<pthread.h>
#include<unistd.h>
#define NUM_READERS 5
#define NUM_WRITERS 2
#define STRING_LENGTH 60
pthread_t readers[NUM_READERS], writers[NUM_WRITERS];
sem_t mutex, rw_mutex;
int readers_count = 0;
FILE *file;
char generateRandomChar() {
return (char)('a' + rand() % 26);
}
```

```c
void *reader(void *arg) {
while (1) {

sem_wait(&mutex);
readers_count++;
if (readers_count == 1) {
sem_wait(&rw_mutex);
}
sem_post(&mutex);

fseek(file, 0, SEEK_SET);
char buffer[256];
while (fgets(buffer, sizeof(buffer), file) != NULL) {
fprintf(stdout,"Reader %ld: %s", (long)arg, buffer);
}
```

```c
sem_wait(&mutex);
readers_count--;
if (readers_count == 0) {
sem_post(&rw_mutex);
}
sem_post(&mutex);

usleep(100000);
}
}
```

```c
void *writer(void *arg) {
while (1) {
sem_wait(&rw_mutex);
srand(time(NULL));
char randomString[STRING_LENGTH + 1];
for (int i = 0; i < STRING_LENGTH; i++) {
randomString[i] = generateRandomChar();
}
randomString[STRING_LENGTH] = '\0';
fseek(file, 0, SEEK_END);
fprintf(file, "%s\n",randomString);
fprintf(stdout, "Writer %ld: %s\n", (long)arg, randomString);
fflush(file);
sem_post(&rw_mutex);

usleep(100000);
}
}
```

```c
int main() {
    file = fopen("shared_file.txt", "a+");
    if (file == NULL) {
    perror("Error opening file");
    exit(EXIT_FAILURE);
    }
    sem_init(&mutex, 0, 1); sem_init(&rw_mutex, 0, 1);
    int i;
    for (i = 0; i < NUM_WRITERS; i++) pthread_create(&writers[i], NULL, writer, (void *)(long)i);
    for (i = 0; i < NUM_READERS; i++) pthread_create(&readers[i], NULL, reader, (void *)(long)i);
    for (i = 0; i < NUM_READERS; i++) pthread_join(readers[i], NULL);
    for (i = 0; i < NUM_WRITERS; i++) pthread_join(writers[i], NULL);
    fprintf(stdout, "reader pthread join completed\n");
    sem_destroy(&mutex);
    sem_destroy(&rw_mutex);
    fclose(file);
    return 0;
}
```

SIGNALS

- Mechanism for **inter-process communication**.
- Used to notify a process of events like errors, termination requests, alarms, etc.
- UNIX sends signals automatically (e.g., on Ctrl+C, Ctrl+Z, division by zero), but processes can send signals too.

2. Types of Signals: Study the most commonly used signals and their default behavior:

- SIGINT – sent on Ctrl+C (terminates)
- SIGTSTP – sent on Ctrl+Z (suspends)
- SIGTERM – standard termination
- SIGKILL – forces termination (cannot be caught or ignored)
- SIGALRM – alarm clock
- SIGSEGV – segmentation fault (invalid memory access)
- SIGFPE – floating point exception
- SIGQUIT, SIGILL, SIGPIPE, etc.

Each signal has:

- A default action (terminate, ignore, suspend)
- Can be **caught** with a signal handler or **ignored**

3. Important System Calls to Learn

Understand and learn how to use the following:

signal()

Used to assign a custom handler to a signal.

```
#include <signal.h>
void handler(int signum) {
   printf("Caught signal %d\n", signum);
}
signal(SIGINT, handler); // Catch Ctrl+C
```

✅ **alarm()**

Sets a timer for delivering SIGALRM.

```
#include <unistd.h>
alarm(5); // After 5 seconds, SIGALRM is sent
```

✅ **pause()**

Suspends the program until a signal is received.

pause(); // Wait until a signal is received

✅ **kill()**

Used to send a signal to another process.

kill(pid, SIGINT); // Sends SIGINT to the given process

✅ **sigaction()**

More reliable and advanced version of signal().

struct sigaction sa;
sa.sa_handler = &handler;
sigaction(SIGINT, &sa, NULL);

4. Critical Code Protection and Signal Chaining

IGNORING SIGNALS:

```c
#include<stdio.h>
#include<unistd.h>
#include<signal.h>

int main() {
    void (*prev_handler)(int);
    printf("SIGINT unignored\n");
    sleep(5);
    prev_handler = signal(SIGINT,SIG_IGN);
    printf("SIGINT ignored\n");
    sleep(5);
    signal(SIGINT,prev_handler);
    printf("SIGINT unignored\n");
    sleep(5);
    return 0;
}
```

```
kinza@DESKTOP-LKI25JK:~$ gcc oslabtasks.c -o out
kinza@DESKTOP-LKI25JK:~$ ./out
SIGINT unignored
SIGINT ignored
^C
^C
SIGINT unignored
^C
```

5. Using SIGACTION to handle interrupts

```c
#include <stdio.h> //needs for perror
#include <signal.h> //signal.h
#include<wait.h>
#include<unistd.h>

void handler(int signum){
if(signum == SIGINT)
{
printf("CONTROL SIGNAL IS PRESSED!");
}
}
int main(){
struct sigaction sa;
//creating sa, which will be called in sigaction function with the control signal variable
sa.sa_handler = handler;
//this is declaring which handler is used if control signal is passed to sa;
while(1){
printf("/");
for(int i=0;i<=100000;i++){
}
if(sigaction(SIGINT, &sa, NULL) == -1)
perror("S1GACTION");
}
return 0;
}
```

```
////////////////////////////////////////////////
////////////////////////////////////////////////
///////////////CONTROL SIGNAL IS PRESSED!//////
////////////////////////////////////////////////
////////////////////////////////////////////////
////////////////////////////////////////////////
////////////////////////////////////////////////
```

LAB 9 ACTIVITY

Task1

Convert the following code of signal into sigaction

```
#include<stdio.h>
#include<signal.h>
#include<unistd.h>
void sig_handler(int signo)
{
if (signo == SIGUSR1)
printf("received SIGUSR1\n");
else if (signo == SIGKILL)
printf("received SIGKILL\n");
else if (signo == SIGSTOP)
printf("received SIGSTOP\n");
}
int main(void)
{
if (signal(SIGUSR1, sig_handler) == SIG_ERR)
printf("\ncan't catch SIGUSR1\n");
if (signal(SIGKILL, sig_handler) == SIG_ERR)
printf("\ncan't catch SIGKILL\n");
if (signal(SIGSTOP, sig_handler) == SIG_ERR)
printf("\ncan't catch SIGSTOP\n");
// A long long wait so that we can easily issue a signal to this
process
while(1)
sleep(1);
return 0;
}
```

```c
#include<stdio.h>
#include<signal.h>
#include<unistd.h>
void sig_handler(int signo)
{
if (signo == SIGUSR1)
printf("received SIGUSR1\n");
else if (signo == SIGKILL)
printf("received SIGKILL\n");
else if (signo == SIGSTOP)
printf("received SIGSTOP\n");
}
int main(void)
{
    struct sigaction sa;
    sa.sa_handler = sig_handler;
    sa.sa_flags = 0;
    sigemptyset(&sa.sa_mask);

if (sigaction(SIGUSR1,&sa,NULL) == -1)
printf("\ncan't catch SIGUSR1\n");
if (sigaction(SIGKILL,&sa,NULL) == -1)
printf("\ncan't catch SIGKILL\n");
if (sigaction(SIGSTOP,&sa,NULL) == -1)
printf("\ncan't catch SIGSTOP\n");
//A long long wait so that we can easily issue a signal to this process
while(1)
sleep(1);
return 0;
}
```

```
kinza@DESKTOP-LKI25JK:~$ gcc oslabtasks.c -o out
kinza@DESKTOP-LKI25JK:~$ ./out

can't catch SIGKILL

can't catch SIGSTOP
^C
```

Task 2

Write a program to ignore SIGKILL and SIGSTOP

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

int main() {
    signal(SIGKILL, SIG_IGN);  // Try to ignore SIGKILL
    signal(SIGSTOP, SIG_IGN);  // Try to ignore SIGSTOP

    while (1) {
        sleep(1);
    }

    return 0;
}
```

Using SIG_IGN (i.e., signal(signum, SIG_IGN)) **can only be used to ignore *some* signals**, but **not SIGKILL or SIGSTOP**, as they are **explicitly uncatchable, unblockable, and unignorable** by the operating system.

```c
void handle_alarm(int sig) {
    printf("Alarm received! Performing scheduled task.\n");
    // Perform scheduled task here
}
int main() {
    signal(SIGALRM, handle_alarm);  // Set up signal handler
    alarm(5);  // Set alarm for 5 seconds
    // Main program continues running
    while (1) {
        // Perform other tasks
    }
    return 0;
}
```

signal(SIGALRM, handle_alarm);: This line sets up the signal handler. It tells the operating system that when the SIGALRM signal is triggered, it should call the handle_alarm function.

- SIGALRM is a predefined constant for the alarm signal in POSIX systems.
- handle_alarm is the function that will be executed when SIGALRM is received.

alarm(5);: This sets an alarm that will trigger after 5 seconds. The alarm() function schedules a SIGALRM signal to be sent to the process after the specified number of seconds. Here, the alarm is set for 5 seconds.

- After 5 seconds, the SIGALRM signal is sent, and the handle_alarm function is invoked.

Task 3

Modify the code from last week's lab of 'Array Sum' using pthread and semaphore such that each process needs to spend 5 seconds in critical section i.e. you will set an alarm for it and then exits the critical section using signal.

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

#define ARRAY_SIZE 10
#define NUM_THREADS 2
int arr[ARRAY_SIZE] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int total_sum = 0;  // Shared variable for the sum
sem_t semaphore;
void handle_alarm(int sig) {
    printf("Thread's critical section time is over. Exiting.\n");
    sem_post(&semaphore);  // Release the semaphore when the alarm is triggered
}
```

```c
void* thread_function(void* arg) {
    int thread_id = *((int*) arg);
    int start_index = thread_id * (ARRAY_SIZE / NUM_THREADS);
    int end_index =  start_index + (ARRAY_SIZE / NUM_THREADS);
    // Set the signal handler for SIGALRM
    signal(SIGALRM, handle_alarm);
    // Set an alarm for 5 seconds
    alarm(5);
    // Wait for the semaphore to enter the critical section
    sem_wait(&semaphore);
    printf("Thread %d entering critical section.\n", thread_id);
    // Critical section: Sum up the part of the array assigned to this thread
    int sum = 0;
    for (int i = start_index; i < end_index; i++) {
        sum += arr[i];
    }
// Update the global sum safely
    total_sum += sum;
    printf("Thread %d calculated sum: %d\n", thread_id, sum);
    // After 5 seconds, the signal handler will release the semaphore and allow the thread to exit
    return NULL;
}
```

```c
int main() {
    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS];
    // Initialize semaphore with 1 resource (binary semaphore, like a mutex)
    sem_init(&semaphore, 0, 1);
    for (int i = 0; i < NUM_THREADS; i++) {
        thread_ids[i] = i;
        pthread_create(&threads[i], NULL, thread_function,(void *)&thread_ids[i]);
    }
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
    printf("Total sum of the array: %d\n", total_sum);
    sem_destroy(&semaphore);
    return 0;
}
```

```
kinza@DESKTOP-LKI25JK:~$ gcc oslabtasks.c -o out
kinza@DESKTOP-LKI25JK:~$ ./out
Thread 0 entering critical section.
Thread 0 calculated sum: 15
Thread's critical section time is over. Exiting.
Thread 1 entering critical section.
Thread 1 calculated sum: 40
Total sum of the array: 55
```

MULTITHREADING

2 ways:

```c
#define ARRAY_SIZE 18
#define NUM_THREADS 3

int arr[ARRAY_SIZE];
void* assignRandom()
{
    for (int i=0;i<ARRAY_SIZE;i++)
    {
        arr[i] = rand() % 100;
    }
    return NULL;
}
```

```c
void *minarray()
{
    int min= arr[0];
    for (int i=1;i<ARRAY_SIZE;i++)
    {
        if (arr[i]<min)
        {
            min = arr[i];
        }
    }
    int *min_ptr=malloc(sizeof(int));
    *min_ptr=min;
    pthread_exit((void *)min_ptr);
}
```

```c
void *maxarray()
{
    int max=arr[0];
    for(int i=1;i<ARRAY_SIZE;i++)
    {
        if(arr[i]>max)
        {
            max=arr[i];
        }
    }
    int *max_ptr=malloc(sizeof(int));
    *max_ptr =max;
    pthread_exit((void *)max_ptr);
}
```

```c
void *sumarray(void *arg)
{
int sum=0;
int tid = *((int *) arg);
int start = tid*(ARRAY_SIZE/NUM_THREADS);
int end =start+(ARRAY_SIZE/NUM_THREADS);
for (int i=start;i<end;i++)
{
    sum+=arr[i];

}
int *sum_ptr = malloc(sizeof(int));
*sum_ptr = sum;
pthread_exit((void*) sum_ptr);
}
```

```c
int main()
{
    pthread_t threads[NUM_THREADS];
    pthread_t thread1;
    void *thread_results[NUM_THREADS];
    int total_sum=0;
    int thread_ids[NUM_THREADS];
    srand(time(NULL));
    pthread_create(&thread1,NULL,assignRandom,NULL);
    pthread_join(thread1,NULL);
```

```c
for(int i=0;i<NUM_THREADS;i++)
    {
        thread_ids[i] = i;
        pthread_create(&threads[i],NULL,sumarray,&thread_ids[i]);
    }
    for(int i=0;i<NUM_THREADS;i++)
    {
        pthread_join(threads[i],&thread_results[i]);
        total_sum+= *((int*) thread_results[i]);
        free(thread_results[i]); // Free the allocated memory for each thr
    }
    printf("Sum of array elements: %d\n",total_sum);
    pthread_t t2;
    void *min_result;
    pthread_create(&t2,NULL,minarray,NULL);
    pthread_join(t2,&min_result);
    printf("Minimum element in array: %d\n", *((int *)min_result));
    free(min_result); // Free the allocated memory for the minimum result
    pthread_t t3;
    void *max_result;
    pthread_create(&t3,NULL,maxarray,NULL);
    pthread_join(t3,&max_result);
    printf("Maximum result in array: %d\n", *((int *)max_result));
    free(max_result);
    pthread_exit(NULL);
    return 0;
}
```

```
kinza@DESKTOP-LKI25JK:~$ gcc pthreadcode.c -o out -lpthread
kinza@DESKTOP-LKI25JK:~$ ./out
Random numbers assigned to array:
79 46 7 77 61 95 84 24 65 55 46 38 77 98 81 90 12 51
Sum of array elements: 1086
Minimum element in array: 7
Maximum result in array: 98
```

```c
#define ARRAY_SIZE 7
#define NUM_THREADS 3
int arr[ARRAY_SIZE];
int avg;
int min,max;
void *avgarr(void *arg)
{
int sum=0;
for(int i=0;i<ARRAY_SIZE;i++)
{
sum += arr[i];
}
avg =(int)sum/ARRAY_SIZE;
pthread_exit(0);
}
```

```c
void *minarr(void *arg)
{
min=arr[0];
for(int i=1;i<ARRAY_SIZE;i++)
{
if(arr[i] < min)
{
min = arr[i];
}
}
pthread_exit(0);
}
void *maxarr(void *arg)
{
max=arr[0];
for(int i=1;i<ARRAY_SIZE;i++)
{
if(arr[i] > max)
{
max = arr[i];
}
}
pthread_exit(0);
}
```

```c
int main(int argc, char *argv[])
{
if(argc!=8)
{
printf("Usage: %s <arrayelements>", argv[0]);
return 1;
}
for(int i=0;i<7;i++)
{
arr[i]=atoi(argv[i+1]);
}
pthread_t workers[3];
pthread_create(&workers[0], NULL, avgarr, NULL);
pthread_create(&workers[1], NULL, minarr, NULL);
pthread_create(&workers[2], NULL, maxarr, NULL);
    for (int i = 0; i < NUM_THREADS; i++)
    {
        pthread_join(workers[i], NULL);
    }

    printf("Average: %d\n", avg);
    printf("Min: %d\n", min);
    printf("Max: %d\n", max);
return 0;
}
```

FIBONACCI

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int *fibonacci;
int n;
void *generate_fibonacci(void *param);
int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <number_of_terms>\n", argv[0]);
        return 1;
    }
    n = atoi(argv[1]);
    if (n <= 0) {
        printf("Please enter a positive integer.\n");
        return 1;
    }
    fibonacci = (int *)malloc(n * sizeof(int));
    if (fibonacci == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }
    pthread_t fib_thread;
    pthread_create(&fib_thread, NULL, generate_fibonacci, NULL);
    pthread_join(fib_thread, NULL);
    printf("Fibonacci sequence: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", fibonacci[i]);
    }
    printf("\n");
    free(fibonacci);
    return 0;
}
```

```c
// Function to generate Fibonacci sequence
void *generate_fibonacci(void *param) {
    if (n > 0) fibonacci[0] = 0;
    if (n > 1) fibonacci[1] = 1;
    for (int i = 2; i < n; i++) {
        fibonacci[i] = fibonacci[i - 1] + fibonacci[i - 2];
    }
    pthread_exit(0);
}
```

```
kinza@DESKTOP-LKI25JK:~$ gcc pthreadcode.c -o out -lpthread
kinza@DESKTOP-LKI25JK:~$ ./out 4
Fibonacci sequence: 0 1 1 2
kinza@DESKTOP-LKI25JK:~$ ./out 10
Fibonacci sequence: 0 1 1 2 3 5 8 13 21 34
```

Prime Nos.

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
void *find_primes(void *param);
int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <number>\n", argv[0]);
        return 1;
    }
    int limit = atoi(argv[1]);
    if (limit < 2) {
        printf("No prime numbers available.\n");
        return 1;
    }
    pthread_t prime_thread;
    pthread_create(&prime_thread, NULL, find_primes,&limit);
    pthread_join(prime_thread, NULL);
    return 0;
}
```

```c
void *find_primes(void *param) {
    int limit = *((int *)param);
    printf("Prime numbers up to %d: ", limit);
    for (int num = 2; num <= limit; num++) {
        bool is_prime = true;
        for (int i = 2; i * i <= num; i++) {
            if (num % i == 0) {
                is_prime = false;
                break;
            }
        }
        if (is_prime) {
            printf("%d ", num);
        }
    }
    printf("\n");
    pthread_exit(0);
}
```

```
kinza@DESKTOP-LKI25JK:~$ gcc pthreadcode.c -o out -lpthread
kinza@DESKTOP-LKI25JK:~$ ./out 10
Prime numbers up to 10: 2 3 5 7
```