


1(a) Hello pthreads — No Synchronization (Race Condition)

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

int counter = 0;

void* func(void* arg) {
    for (int i = 0; i < 1000; i++)
        counter++;
    return NULL;
}

int main() {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, func, NULL);
    pthread_create(&t2, NULL, func, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Counter = %d\n", counter);
}
```

 *Race condition:* output may vary each time.

1(b) Mutual Exclusion using Mutex

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

int counter = 0;
pthread_mutex_t lock;

void* func(void* arg) {
    for (int i = 0; i < 1000; i++) {
        pthread_mutex_lock(&lock);
        counter++;
        pthread_mutex_unlock(&lock);
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;
    pthread_mutex_init(&lock, NULL);
    pthread_create(&t1, NULL, func, NULL);
    pthread_create(&t2, NULL, func, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Counter = %d\n", counter);
    pthread_mutex_destroy(&lock);
}
```

✓ *Output always correct (no race).*

2. Argumentative pthreads (Multiple counters, argument passing, array of mutex)

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

int data[3];
pthread_mutex_t lock[3];

void* add(void* arg) {
    int i = *(int*)arg;
    for (int j = 0; j < 1000; j++) {
        pthread_mutex_lock(&lock[i]);
        data[i]++;
        pthread_mutex_unlock(&lock[i]);
    }
    return NULL;
}

int main() {
    pthread_t t[3]; int id[3];
    for (int i = 0; i < 3; i++) {
        data[i] = 0;
        pthread_mutex_init(&lock[i], NULL);
        id[i] = i;
        pthread_create(&t[i], NULL, add, &id[i]);
    }
    for (int i = 0; i < 3; i++)
        pthread_join(t[i], NULL);

    for (int i = 0; i < 3; i++)
        printf("data[%d]=%d\n", i, data[i]);
}
```

✓ Each data value safely updated with its own lock.

❶ Reader–Writer Problem (Simple Version)

➡ Using **mutex** — multiple readers can read, one writer writes.

```
#include <stdio.h>      // for printf

#include <pthread.h>     // for pthread functions

#include <unistd.h>      // for sleep


pthread_mutex_t wrt, mutex; // two mutex locks

int readcount = 0, data = 0; // shared counters and data variable


// ----- Reader Function -----

void* reader(void* arg) {

    int id = *(int*)arg; // get reader ID

    while (1) {

        pthread_mutex_lock(&mutex);    // lock to update readcount safely

        readcount++;                  // one more reader enters

        if (readcount == 1)           // first reader locks writer

            pthread_mutex_lock(&wrt);

        pthread_mutex_unlock(&mutex);  // unlock for others


        printf("Reader %d reads data = %d\n", id, data); // reading section

        sleep(1);                      // simulate reading time


        pthread_mutex_lock(&mutex);    // before leaving, lock again
```

```

        readcount--;          // one reader leaves

        if (readcount == 0)    // last reader unlocks writer

            pthread_mutex_unlock(&wrt);

        pthread_mutex_unlock(&mutex);    // unlock for others

        sleep(1);

    }

}

// ----- Writer Function -----

void* writer(void* arg) {

    int id = *(int*)arg;        // get writer ID

    while (1) {

        pthread_mutex_lock(&wrt);    // lock to get exclusive access

        data++;                    // write (update) data

        printf("Writer %d writes data = %d\n", id, data);

        pthread_mutex_unlock(&wrt);    // release access

        sleep(2);                  // simulate writing time

    }

}

// ----- Main Function -----

int main() {

    pthread_t r1, r2, w1;          // 2 readers, 1 writer threads

    int id1=1, id2=2, id3=1;

```

```
pthread_mutex_init(&mutex,NULL);    // initialize mutex
pthread_mutex_init(&wrt,NULL);      // initialize write lock
pthread_create(&r1,NULL,reader,&id1); // create reader thread 1
pthread_create(&r2,NULL,reader,&id2); // create reader thread 2
pthread_create(&w1,NULL,writer,&id3); // create writer thread

pthread_join(r1,NULL);              // wait for threads
pthread_join(r2,NULL);
pthread_join(w1,NULL);
}
```

✅ Output alternates between readers reading and writer writing.

💡 *Very simple, easy to explain in viva.*

2 Producer–Consumer Problem (Simple & Commented)

```
#include <stdio.h>           // for printf
#include <pthread.h>         // for pthread functions
#define SIZE 5              // buffer size

int buffer[SIZE], count=0;   // shared buffer and counter
pthread_mutex_t lock;        // mutex for critical section
pthread_cond_t notFull, notEmpty; // condition variables

// ----- Producer Function -----
void* producer(void* a) {
    int item = 1;            // item number to produce
    while (1) {
        pthread_mutex_lock(&lock);    // enter critical section
        while (count == SIZE)         // if buffer full
            pthread_cond_wait(&notFull, &lock); // wait for consumer
        buffer[count++] = item;       // add item to buffer
        printf("Produced %d\n", item++);
        pthread_cond_signal(&notEmpty); // signal consumer
        pthread_mutex_unlock(&lock);   // exit critical section
    }
}

// ----- Consumer Function -----
void* consumer(void* a) {
    while (1) {
        pthread_mutex_lock(&lock);    // enter critical section
        while (count == 0)            // if buffer empty
            pthread_cond_wait(&notEmpty, &lock); // wait for producer
        printf("Consumed %d\n", buffer[--count]); // remove last item
        pthread_cond_signal(&notFull); // signal producer
        pthread_mutex_unlock(&lock);   // exit critical section
    }
}

// ----- Main Function -----
int main() {
    pthread_t p, c;                // threads
    pthread_mutex_init(&lock, NULL); // init mutex
    pthread_cond_init(&notFull, NULL); // init condition
```

```
pthread_cond_init(&notEmpty, NULL);

pthread_create(&p, NULL, producer, NULL); // create producer
pthread_create(&c, NULL, consumer, NULL); // create consumer

pthread_join(p, NULL);           // wait for threads
pthread_join(c, NULL);
}
```

Use:

- Producer waits when buffer full.
 - Consumer waits when buffer empty.
 - Prevents race condition in shared buffer.
-

3. Dining Philosopher Problem (Simple & Commented)

```
#include <stdio.h>          // for printf

#include <pthread.h>         // for pthread functions

#include <unistd.h>          // for sleep

#define N 5                 // number of philosophers

pthread_mutex_t chopstick[N]; // one mutex per chopstick


// ----- Philosopher Function -----

void* philosopher(void* arg) {

    int id = *(int*)arg;    // philosopher ID

    while (1) {

        printf("Philosopher %d thinking\n", id);

        sleep(1);           // thinking

        pthread_mutex_lock(&chopstick[id]);    // pick left chopstick

        pthread_mutex_lock(&chopstick[(id+1)%N]); // pick right chopstick

        printf("Philosopher %d eating\n", id);

        sleep(1);           // eating time

        pthread_mutex_unlock(&chopstick[id]);    // put down left chopstick

        pthread_mutex_unlock(&chopstick[(id+1)%N]); // put down right chopstick

    }

}
```

```
// ----- Main Function -----

int main() {

    pthread_t p[N];          // 5 philosopher threads

    int id[N];

    for (int i=0;i<N;i++) {

        id[i]=i;              // philosopher number

        pthread_mutex_init(&chopstick[i],NULL); // initialize each chopstick

        pthread_create(&p[i],NULL,philosopher,&id[i]); // create thread

    }

    for (int i=0;i<N;i++)

        pthread_join(p[i],NULL); // wait for threads

}
```

Use:

- Each philosopher picks left and right chopstick to eat.
- Mutex prevents two philosophers from using same chopstick at once.
(*This basic version can cause deadlock but is perfect for simple understanding.*)

Shell Script CommandLine Argument

```
#!/bin/bash
# This script performs basic arithmetic using command-line arguments

# $1 and $2 are the first and second command-line arguments

# Check if exactly two arguments are given
if [ $# -ne 2 ]
then
    echo "Usage: $0 num1 num2" # $0 is the script name
    exit 1                     # exit if arguments missing
fi

# Read the numbers
a=$1
b=$2

# Perform calculations
sum=$((a + b)) # addition
diff=$((a - b)) # subtraction
prod=$((a * b)) # multiplication

# Check division by zero
if [ $b -ne 0 ]
then
    div=$((a / b)) # division
else
    div="undefined (division by zero)"
fi

# Display results
echo "First number: $a"
echo "Second number: $b"
echo "Sum: $sum"
echo "Difference: $diff"
echo "Product: $prod"
echo "Quotient: $div"
```

