WORKSHOP 4
PART 1
PratikDhimal_2407779

1.penny01

```
void add_penny(int *balance) {
  int b = *balance;
  usleep(1000000);
  b = b + 1;
  *balance = b;
}

int main(){
  int account = 0;
  add_penny(&account);
  printf("accumulated %dp\n", account);
  return 0;
}
pratikdhimal@resman-hpc:~/hpc/week4/penny-adder$
```

Here, we have created an add_penny function that takes in the memory address of balance and then adds one to the balance and then again sets the balance to the memory address. Now when we call the function the actual memory address's variable will be change and the output will be:

accumulated 1.

2. penny02

```
#include <stdio.h>
#include <unistd.h>
#include "time_diff.h"

void add_penny(int *balance) {
  int b = *balance;
  usleep(1000000);
  b = b + 1;
  *balance = b;
}

int main(){
  struct timespec start, finish;
  long long int difference;
  int account = 0;
  clock_gettime(CLOCK_MONOTONIC, &start);

  add_penny(&account);

  clock_gettime(CLOCK_MONOTONIC, &finish);
  time_difference(&start, &finish, &difference);
  printf("accumulated %dp\n", account);
  printf("run lasted %9.5lfs\n", difference/1000000000.0);
  return 0;
}
pratikdhimal@resman-hpc:~/hpc/week4/penny-adder$ ./penny02
accumulated 1p
run lasted   1.00010s
```

Here the logic is completely same to penny01 but we have added a time calculation process. We use the clock_gettime() function to store the start and end time and lastly the time_difference() function calculates the time taken to execute the code. In this case the time is 1.00010s

3. penny03

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

void add_penny(int *balance) {
  int b = *balance;

// 1 second delay (simulating large calculation time)

  usleep(1000000);

  b = b + 1;
  *balance = b;
}

int main(){
  struct timespec start, finish;
  int i;
  long long int difference;
  int account = 0;
  clock_gettime(CLOCK_MONOTONIC, &start);

  for(i=0;i<5;i++){
    add_penny(&account);
  }

  clock_gettime(CLOCK_MONOTONIC, &finish);
  time_difference(&start, &finish, &difference);

  printf("accumulated %dp\n", account);
  printf("run lasted %lldns or %9.5lfs\n", difference, difference/1000000000.0);
  return 0;
}
pratikdhimal@resman-hpc:~/hpc/week4/penny-adder$ ./penny03
accumulated 5p
run lasted 5000487515ns or   5.00049s
pratikdhimal@resman-hpc:~/hpc/week4/penny-adder$
```

Similar to penny 2 we have created a function to add penny to the balance variable but here we have done it in a loop so that the penney keeps on adding for 5 times hence the balance becomes 5, and the time taken here in thai case is around 5 sec which is justifiable.

4. penney 04

```
#include <stdio.h>
#include <stdlib.h>
#include "time_diff.h"
#include <pthread.h>
#include <unistd.h>

void *add_penny(void *balance) {
  int *b = balance;
  int c = *b;

// 1 second delay (simulating large calculation time)

  usleep(1000000);

  c = c + 1;
  *b = c;
}

int main(){
  struct timespec start, finish;
  long long int difference;
  int account = 0;

  clock_gettime(CLOCK_MONOTONIC, &start);

  pthread_t t;

  /* start a thread to call the add_penny function */
  void *add_penny();
  pthread_create(&t, NULL, add_penny, &account);

  /* wait for the thread to finish*/
  pthread_join(t, NULL);

  clock_gettime(CLOCK_MONOTONIC, &finish);
  time_difference(&start, &finish, &difference);
  printf("accumulated %dp\n", account);
  printf("run lasted %lldns or %9.5lfs\n", difference, difference/1000000000.0);
  return 0;
}
pratikdhimal@resman-hpc:~/hpc/week4/penny-adder$ ./penny04
accumulated 1p
run lasted 1000203323ns or   1.00020s
pratikdhimal@resman-hpc:~/hpc/week4/penny-adder$
```

Similarly, we have calculated the time to execute the program but the only difference here is instead of just adding the penney and assigning it to the balance variable what we have done is created a new variable c and added the memory address to the balance variable. **Same same but different.**

5.penney 05

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

int main(){
  struct timespec start, finish;
  long long int difference;
  int account = 0;
  int i;

  int n = 5;

  clock_gettime(CLOCK_MONOTONIC, &start);

  void *add_penny();
  pthread_t t[n];
  for(i=0;i<n;i++){
    pthread_create(&t[i], NULL, add_penny, &account);
  }
  for(i=0;i<n;i++){
    pthread_join(t[i], NULL);
  }

  clock_gettime(CLOCK_MONOTONIC, &finish);
  time_difference(&start, &finish, &difference);
  printf("accumulated %dp\n", account);
  printf("run lasted %lldns or %9.5lfs\n", difference, difference/1000000000.0);
  return 0;
}

void *add_penny(int *balance) {
  int b = *balance;
  usleep(1000000);
  b = b + 1;
  *balance = b;
}
```

```
pratikdhimal@resman-hpc:~/hpc/week4/penny-adder$ ./penny05
accumulated 1p
run lasted 1000373363ns or   1.00037s
pratikdhimal@resman-hpc:~/hpc/week4/penny-adder$
```

Here 5 threads are trying to update the penny but due to race conditions the value we get is not what we want. We'll solve this problem in penney 08.

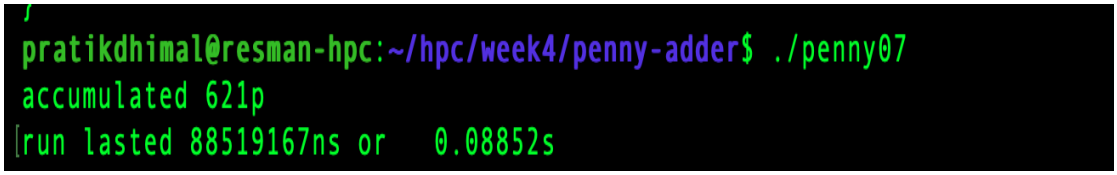
6. penney 06

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

int main(){
  struct timespec start, finish;
  long long int difference;
  int account = 0;
  int i;

  int n = 100;

  clock_gettime(CLOCK_MONOTONIC, &start);

  void *add_penny();
  pthread_t t[n];
  for(i=0;i<n;i++){
    pthread_create(&t[i], NULL, add_penny, &account);
  }
  for(i=0;i<n;i++){
    pthread_join(t[i], NULL);
  }

  clock_gettime(CLOCK_MONOTONIC, &finish);
  time_difference(&start, &finish, &difference);
  printf("accumulated %dp\n", account);
  printf("run lasted %lldns or %9.5lfs\n", difference, difference/1000000000.0);
  return 0;
}

void *add_penny(int *balance) {
  int b = *balance;
/* cause a short delay */
  int i;
  for(i=0;i<100000;i++){
  }
  b = b + 1;
  *balance = b;
}
```

```
pratikdhimal@resman-hpc:~/hpc/week4/penny-adder$ ./penny06
accumulated 58p
run lasted 8958288ns or   0.00896s
```

Here the logic is very similar to penney 5, then difference is we have added 100 threads instead of 5 and we have created a loop to loop to 10000. Hence, creating the delay. The race condition is still prevalent in this case.

7. penney 07



This program creates 1000 threads, and each one tries to add 1 penny to the shared variable account. Because the threads update the shared variable without locking the mutex inside add_penny(), they interfere with each other, causing lost updates. As a result, the total is incorrect (621 instead of 1000) due to race conditions. The program also measures how long the entire threaded execution takes using clock_gettime(). After all threads finish and the time is printed, the program cleans up the mutex and exits.

8.penney 08

```
pratikdhimal@resman-hpc:~/hpc/week4/penny-adder$ ./penny08
accumulated 1000p
run lasted 95237841ns or   0.09524s
pratikdhimal@resman-hpc:~/hpc/week4/penny-adder$
```

Finally, in this code the actual value of balance is updated as we have used mutex to solve the problem of race condition. When a thread adds that balance lock is acquired and other threads must wait until the thread function which is: addPenney() releases the lock. So by thai way it synchronizes the tasks and gives the real count . One thing to consider is that thai takes a bit more time than our previous as threads need to wait before acquiring the lock so that the previous thread completes the task.

9. Penney 09

```
pratikdhimal@resman-hpc:~/hpc/week4/penny-adder$ ./penny09
one
two
one
one
one
one
one
one
one
one
one
three
two
three
two
three
two
three
two
three
two
three
two
three
two
three
two
three
two
three
two
three
accumulated 10p
run lasted 10002084527ns or  10.00208s
pratikdhimal@resman-hpc:~/hpc/week4/penny-adder$
```

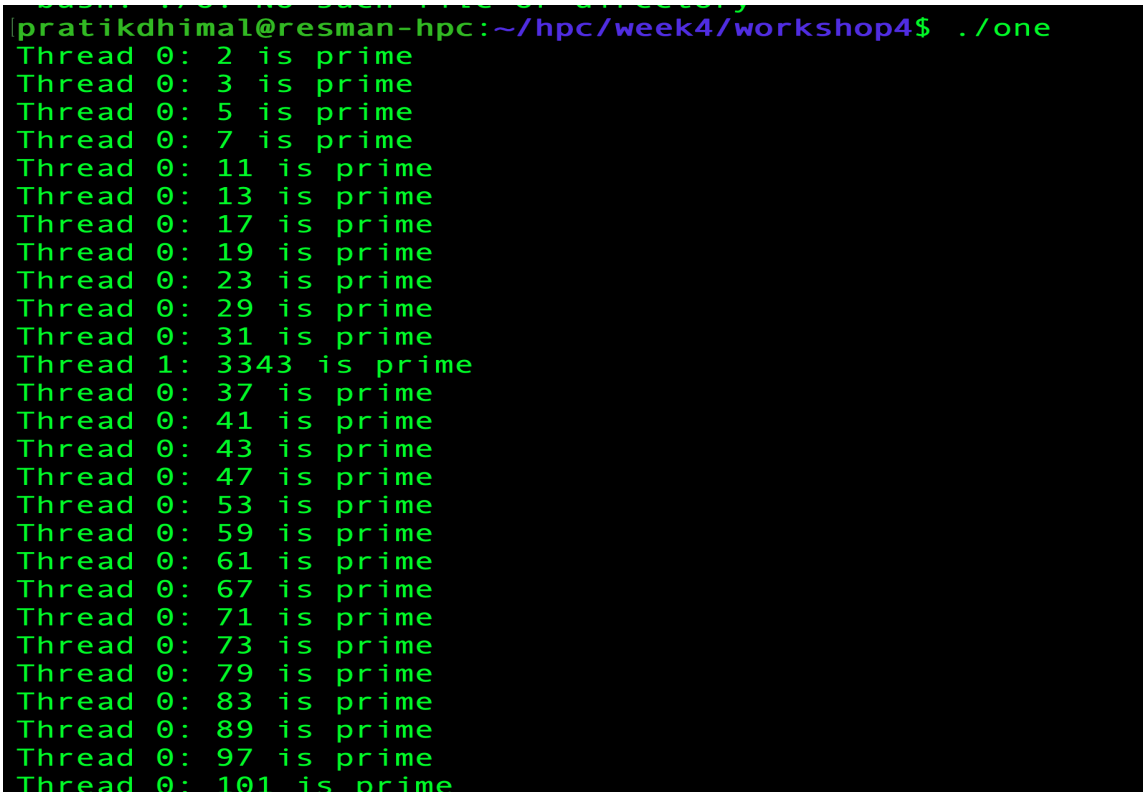
This program creates 10 threads, and each thread runs add_penny() to safely increase the shared account variable. Inside add_penny(), each thread prints messages ("one", "two", "three") and then uses pthread_mutex_lock() to ensure only one thread can update the balance at a time. A (usleep(1000000)) is added to make the locking behavior easy to observe changes. Because the mutex protects the critical section, all 10 increments

happen correctly **without race conditions**. After all threads finish, the program prints the final account total and the time taken before destroying the mutex.

WORKSHOP 4
PART2

1. one.c

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

int is_prime(int n) {
    if (n <= 1) return 0;
    if (n <= 3) return 1;
    if (n % 2 == 0 || n % 3 == 0) return 0;
    for (int i = 5; i * i <= n; i += 6)
        if (n % i == 0 || n % (i + 2) == 0) return 0;
    return 1;
}

void *find_primes(void *arg) {
    int id = *(int *)arg;
    int start = 1 + id * 3334;
    int end = (id == 2) ? 10001 : start + 3334;
    for (int i = start; i < end; i++)
        if (is_prime(i))
            printf("Thread %d: %d is prime\n", id, i);
    return NULL;
}

int main() {
    pthread_t t[3];
    int ids[3] = {0, 1, 2};
    for (int i = 0; i < 3; i++)
        pthread_create(&t[i], NULL, find_primes, &ids[i]);
    for (int i = 0; i < 3; i++)
        pthread_join(t[i], NULL);
    return 0;
}
"one.c" 32L, 796B
```



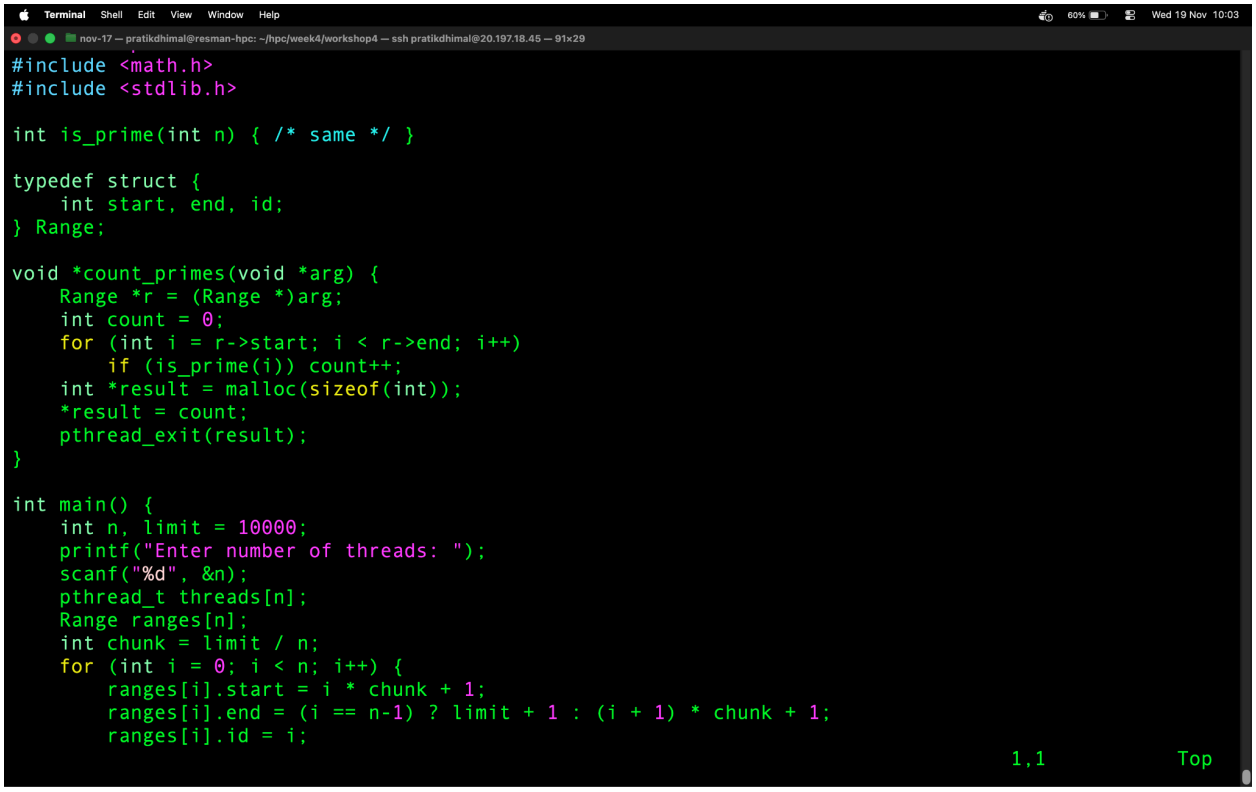
The range 1–10000 is divided into three almost-equal parts. Each of the three threads checks its own segment for prime numbers using an efficient trial division method. Threads run concurrently and print primes as they find them.

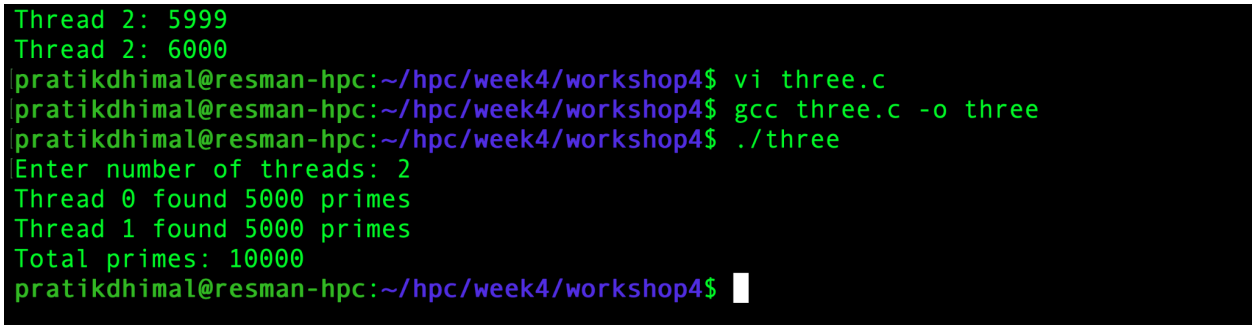
2. two.c

```c
#include <stdlib.h>

int is_prime(int n) { /* same as above */ }

typedef struct {
    int start, end, thread_id;
} Range;

void *find_primes(void *arg) {
    Range *r = (Range *)arg;
    for (int i = r->start; i < r->end; i++)
        if (is_prime(i))
            printf("Thread %d: %d\n", r->thread_id, i);
    return NULL;
}

int main() {
    int n, limit = 10000;
    printf("Enter number of threads: ");
    scanf("%d", &n);
    pthread_t threads[n];
    Range ranges[n];
    int chunk = limit / n;
    for (int i = 0; i < n; i++) {
        ranges[i].start = i * chunk + 1;
        ranges[i].end = (i == n-1) ? limit + 1 : (i + 1) * chunk + 1;
        ranges[i].thread_id = i;
        pthread_create(&threads[i], NULL, find_primes, &ranges[i]);
    }
    for (int i = 0; i < n; i++) pthread_join(threads[i], NULL);
    return 0;
}
```

The program asks the user how many threads to create. It then divides the range 1–10000 equally among the requested threads. Each thread receives a struct containing its start, end, and ID, then prints primes in its range.

3. Three.c





Each thread now counts primes instead of printing them. It allocates an integer, stores the count, and exits using pthread_exit(). The main thread joins each thread, retrieves the count, prints it per thread, and sums them.

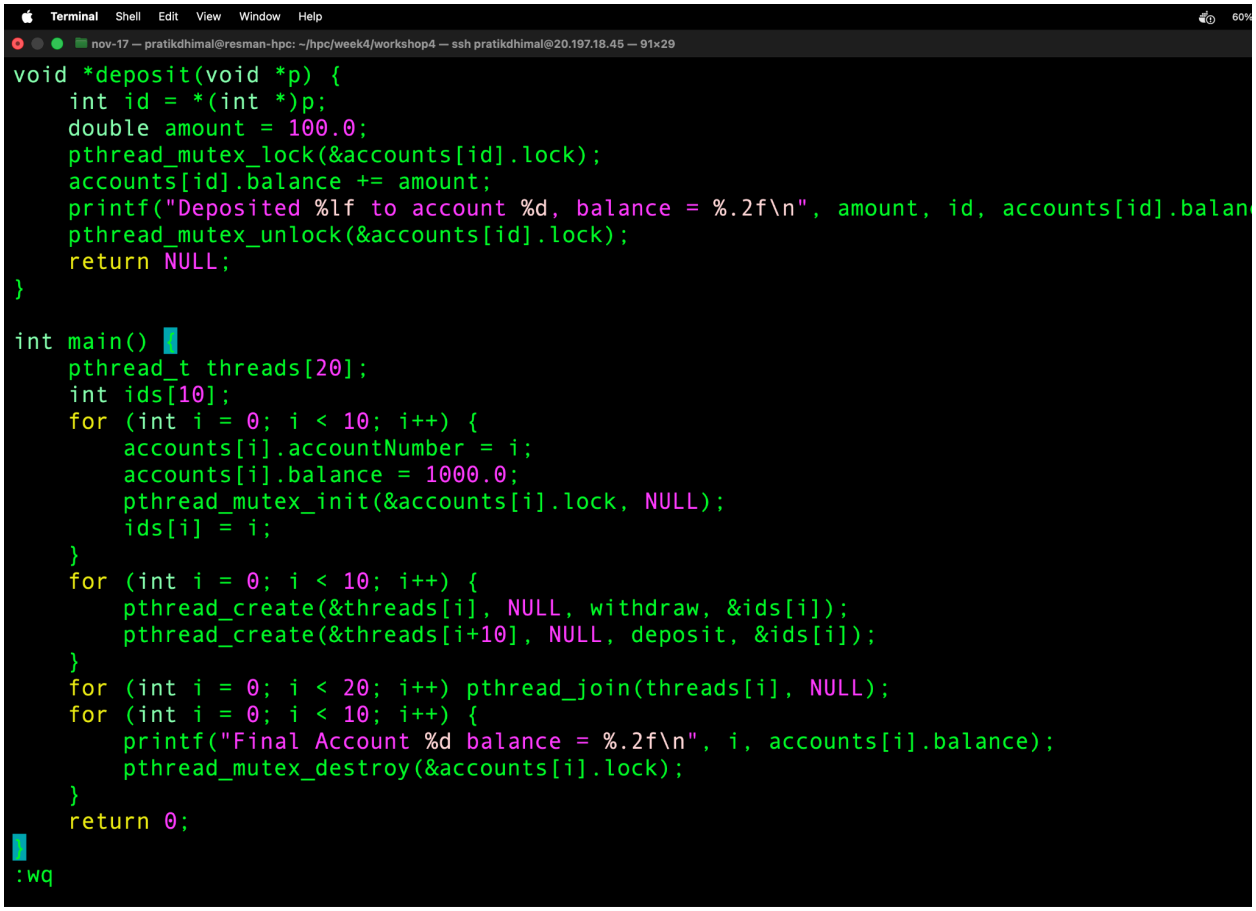
4. Four.c

```c
#include <stdlib.h>
#include <stdatomic.h>

_Atomic int prime_count = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int is_prime(int n) { /* same */ }

void *find_and_count(void *arg) {
    int start = *(int *)arg;
    for (int i = start; i <= 10000; i += 8) {  // 8 threads max safe step
        if (is_prime(i)) {
            pthread_mutex_lock(&mutex);
            prime_count++;
            if (prime_count >= 5) {
                pthread_mutex_unlock(&mutex);
                return NULL;
            }
            printf("Found prime %d: %d\n", prime_count, i);
            pthread_mutex_unlock(&mutex);
        }
    }
    return NULL;
}

int main() {
    int n;
    printf("Enter number of threads: ");
    scanf("%d", &n);
    pthread_t threads[n];
    int starts[n];
    for (int i = 0; i < n; i++) {
```

```
    |                                  ^~~~~
pratikdhimal@resman-hpc:~/hpc/week4/workshop4$ ./four
Enter number of threads: 5
Found prime 1: 2
Found prime 2: 10
Found prime 3: 18
Found prime 4: 26
5th prime found, all threads cancelled. Total found: 9
pratikdhimal@resman-hpc:~/hpc/week4/workshop4$
```

A global atomic counter tracks how many primes have been found. As soon as any thread discovers the 5th prime, the main thread detects this and immediately cancels all running threads using pthread_cancel(). Remaining threads stop instantly.

5. five .c


Terminal Shell Edit View Window Help
nov-17 — pratikdhimal@resman-hpc: ~/hpc/week4/workshop4 — ssh pratikdhimal@20.197.18.45 — 91×29
void *deposit(void *p) {
int id = *(int *)p;
double amount = 100.0;
pthread_mutex_lock(&accounts[id].lock);
accounts[id].balance += amount;
printf("Deposited %lf to account %d, balance = %.2f\n", amount, id, accounts[id].balan
pthread_mutex_unlock(&accounts[id].lock);
return NULL;
}
int main() {
pthread_t threads[20];
int ids[10];
for (int i = 0; i < 10; i++) {
accounts[i].accountNumber = i;
accounts[i].balance = 1000.0;
pthread_mutex_init(&accounts[i].lock, NULL);
ids[i] = i;
}
for (int i = 0; i < 10; i++) {
pthread_create(&threads[i], NULL, withdraw, &ids[i]);
pthread_create(&threads[i+10], NULL, deposit, &ids[i]);
}
for (int i = 0; i < 20; i++) pthread_join(threads[i], NULL);
for (int i = 0; i < 10; i++) {
printf("Final Account %d balance = %.2f\n", i, accounts[i].balance);
pthread_mutex_destroy(&accounts[i].lock);
}
return 0;
}
:wq

```
ratikdhimal@resman-hpc:~/hpc/week4/workshop4$ gcc five.c -o five
ratikdhimal@resman-hpc:~/hpc/week4/workshop4$ ./five
Withdrew 100.000000 from account 0, balance = 900.00
Withdrew 100.000000 from account 1, balance = 900.00
Deposited 100.000000 to account 0, balance = 1000.00
Deposited 100.000000 to account 1, balance = 1000.00
Withdrew 100.000000 from account 2, balance = 900.00
Deposited 100.000000 to account 2, balance = 1000.00
Withdrew 100.000000 from account 3, balance = 900.00
Deposited 100.000000 to account 3, balance = 1000.00
Withdrew 100.000000 from account 7, balance = 900.00
Withdrew 100.000000 from account 5, balance = 900.00
Deposited 100.000000 to account 7, balance = 1000.00
Deposited 100.000000 to account 5, balance = 1000.00
Withdrew 100.000000 from account 4, balance = 900.00
Withdrew 100.000000 from account 6, balance = 900.00
Deposited 100.000000 to account 4, balance = 1000.00
Withdrew 100.000000 from account 8, balance = 900.00
Deposited 100.000000 to account 6, balance = 1000.00
Deposited 100.000000 to account 8, balance = 1000.00
Withdrew 100.000000 from account 9, balance = 900.00
Deposited 100.000000 to account 9, balance = 1000.00
Final Account 0 balance = 1000.00
Final Account 1 balance = 1000.00
Final Account 2 balance = 1000.00
Final Account 3 balance = 1000.00
Final Account 4 balance = 1000.00
Final Account 5 balance = 1000.00
Final Account 6 balance = 1000.00
Final Account 7 balance = 1000.00
Final Account 8 balance = 1000.00
Final Account 9 balance = 1000.00
ratikdhimal@resman-hpc:~/hpc/week4/workshop4$
```

Each account has its own mutex. Before any deposit or withdrawal, the thread locks that account's mutex, performs the update safely, then unlocks. This prevents race conditions even when multiple threads access the same account concurrently.

6. Six.c

```
inal Account 9 balance = 1000.00
ratikdhimal@resman-hpc:~/hpc/week4/workshop4$ vi six.c
ratikdhimal@resman-hpc:~/hpc/week4/workshop4$ cat six.c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

em_t printers;

oid *print_document(void *arg) {
    int user = *(int *)arg;
    printf("User %d wants to print\n", user);
    sem_wait(&printers);
    printf(">>> User %d is printing...\n", user);
    sleep(2);  // printing time
    printf("<<< User %d finished printing\n", user);
    sem_post(&printers);
    return NULL;


nt main() {
    sem_init(&printers, 0, 2);  // only 2 printers available
    pthread_t users[10];
    int ids[10] = {1,2,3,4,5,6,7,8,9,10};
    for (int i = 0; i < 10; i++)
        pthread_create(&users[i], NULL, print_document, &ids[i]);
    for (int i = 0; i < 10; i++)
        pthread_join(users[i], NULL);
    sem_destroy(&printers);
    return 0;
```

```
pratikdhimal@resman-hpc:~/hpc/week4/workshop4$ ./six
User 1 wants to print
>>> User 1 is printing...
User 2 wants to print
>>> User 2 is printing...
User 3 wants to print
User 4 wants to print
User 6 wants to print
User 5 wants to print
User 7 wants to print
User 8 wants to print
User 9 wants to print
User 10 wants to print
<<< User 1 finished printing
>>> User 3 is printing...
<<< User 2 finished printing
>>> User 4 is printing...
<<< User 3 finished printing
>>> User 6 is printing...
<<< User 4 finished printing
>>> User 5 is printing...
<<< User 6 finished printing
>>> User 7 is printing...
<<< User 5 finished printing
>>> User 8 is printing...
<<< User 7 finished printing
<<< User 8 finished printing
>>> User 9 is printing...
>>> User 10 is printing...
<<< User 9 finished printing
<<< User 10 finished printing
pratikdhimal@resman-hpc:~/hpc/week4/workshop4$
```

A semaphore is initialized to 2 representing two available printers. Each user thread calls sem_wait() to acquire a printer; if both are busy, it blocks. After printing, sem_post() releases the printer so another user can print. Maximum 2 users print simultaneously.