#include <pthread.h>

#include <semaphore.h>

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <unistd.h>

* **pthread.h**: Provides threading functionality for creating and managing threads.
* **semaphore.h**: Used for semaphores, a synchronization primitive to prevent deadlocks or race conditions.
* **stdio.h**: Provides functions for input/output operations like printf.
* **stdlib.h**: Includes functions for dynamic memory management (malloc, free) and conversions (atoi).
* **time.h**: Allows working with time-related functions, such as generating random numbers.
* **unistd.h**: Provides access to the usleep function to introduce delays in microseconds.

**2. Define Structures**

c

Copy code

struct Philosopher {

int number;

int leftForkIndex;

int rightForkIndex;

int eatenTimes;

pthread\_t thread\_id;

};

* **number**: Identifier for each philosopher.
* **leftForkIndex/rightForkIndex**: Index of the forks assigned to the philosopher.
* **eatenTimes**: Tracks how many times the philosopher has eaten.
* **thread\_id**: Stores the thread ID for the philosopher's thread.

c

Copy code

struct Fork {

int index;

sem\_t mutex;

};

* **index**: Identifier for the fork.
* **mutex**: Semaphore to lock/unlock the fork to ensure only one philosopher uses it at a time.

**3. Global Variables**

c

Copy code

struct Fork \*forks;

sem\_t global\_mutex;

int NotEatenCount = 0;

* **forks**: Pointer to an array of Fork structures.
* **global\_mutex**: Global semaphore to synchronize access to shared variables like NotEatenCount.
* **NotEatenCount**: Tracks the number of philosophers who haven't eaten yet.

**4. Function: is\_finished**

c

Copy code

int is\_finished() {

int counter = 0;

sem\_wait(&global\_mutex);

counter = NotEatenCount;

sem\_post(&global\_mutex);

return counter == 0;

}

* Locks the global mutex to read NotEatenCount safely.
* Returns 1 (true) if all philosophers have eaten at least once (NotEatenCount == 0); otherwise, returns 0 (false).

**5. Thread Function: philosopher\_thread**

c

Copy code

void \*philosopher\_thread(void \*argument) {

struct Philosopher \*philosopher = (struct Philosopher \*)argument;

int again = 1;

* **argument**: The philosopher passed to the thread.
* **again**: Used to control whether the philosopher continues attempting to eat.

**Philosopher Thinking**

c

Copy code

printf("Philosopher %d is Thinking\n", philosopher->number);

usleep(500 \* (1000 + 100 \* (rand() % 60)));

* Logs that the philosopher is thinking.
* Introduces a randomized delay between 0.5 to 3.5 seconds to simulate thinking.

**Philosopher Attempts to Eat**

c

Copy code

printf("Philosopher %d is trying to eat...\n", philosopher->number);

if (sem\_trywait(&forks[philosopher->leftForkIndex].mutex) == 0) {

* The philosopher tries to lock the left fork semaphore. If successful, they proceed to try for the right fork.

**Waiting for the Right Fork**

c

Copy code

int waiting\_times = 10 + rand() % 50;

while (waiting\_times > 0) {

if (sem\_trywait(&forks[philosopher->rightForkIndex].mutex) == 0) {

* Introduces a randomized waiting time (between 10–59 cycles) for acquiring the right fork.
* If successful, logs that the philosopher is eating.

**Eating**

c

Copy code

printf("Philosopher %d is Eating\n", philosopher->number);

if (!philosopher->eatenTimes) {

sem\_wait(&global\_mutex);

NotEatenCount--;

sem\_post(&global\_mutex);

}

philosopher->eatenTimes++;

usleep(500 \* (1000 + 100 \* (rand() % 60)));

sem\_post(&forks[philosopher->rightForkIndex].mutex);

* Logs eating and decrements NotEatenCount if the philosopher eats for the first time.
* Introduces a delay to simulate eating, then releases the right fork semaphore.

**Failure to Acquire Fork**

c

Copy code

waiting\_times--;

usleep(100000);

if (waiting\_times == 0) {

printf("Philosopher %d cannot take second fork...\n", philosopher->number);

}

sem\_post(&forks[philosopher->leftForkIndex].mutex);

* If the philosopher can't acquire the right fork within the allowed time, they release the left fork and retry later.

**Check if Finished**

c

Copy code

again = !is\_finished();

* The philosopher checks if all philosophers have eaten at least once.

**6. Main Function**

**Parse Command-line Arguments**

c

Copy code

int main(int argc, char \*argv[]) {

struct Philosopher \*philosophers;

int i, count = 5;

if (argc >= 2) count = atoi(argv[1]);

srand((unsigned int)time(NULL));

if (count < 2 || count > 1000) count = 5;

* Reads the number of philosophers from the command line. If invalid, defaults to 5.
* Seeds the random number generator for randomized delays.

**Allocate Memory**

c

Copy code

philosophers = (struct Philosopher \*)malloc(sizeof(struct Philosopher) \* count);

forks = (struct Fork \*)malloc(sizeof(struct Fork) \* count);

sem\_init(&global\_mutex, 0, 1);

NotEatenCount = count;

* Dynamically allocates memory for philosophers and forks.
* Initializes the global mutex semaphore and sets NotEatenCount.

**Initialize Philosophers and Forks**

c

Copy code

for (i = 0; i < count; i++) {

sem\_init(&forks[i].mutex, 0, 1);

philosophers[i].eatenTimes = 0;

philosophers[i].number = i + 1;

philosophers[i].leftForkIndex = i;

philosophers[i].rightForkIndex = (i + 1) % count;

}

* Initializes fork semaphores.
* Sets up each philosopher with their ID, left and right fork indices, and initializes their eaten count to 0.

**Create Threads**

c

Copy code

for (i = 0; i < count; i++)

pthread\_create(&philosophers[i].thread\_id, NULL, philosopher\_thread, &philosophers[i]);

* Creates a thread for each philosopher, passing their data to the philosopher\_thread function.

**Wait for Threads to Finish**

c

Copy code

while (!is\_finished()) usleep(100);

for (i = 0; i < count; i++)

pthread\_join(philosophers[i].thread\_id, NULL);

* Continuously checks if all philosophers have eaten.
* Joins all threads after completion.

**Output Statistics and Clean Up**

c

Copy code

printf("\nStatistics:\n");

for (i = 0; i < count; i++) {

printf("Philosopher %d eaten for %d times\n", philosophers[i].number, philosophers[i].eatenTimes);

}

free(forks);

free(philosophers);

return 0;

* Logs how many times each philosopher has eaten.
* Frees dynamically allocated memory.

**Purpose of the Code**

The code simulates the **Dining Philosophers Problem**, a classic problem in concurrency. It ensures:

1. **Deadlock Prevention**: By using sem\_trywait and timed retries.
2. **Starvation Avoidance**: All philosophers eat at least once.
3. **Concurrency Control**: Forks are shared among threads using semaphores.

This demonstrates efficient synchronization mechanisms using threads and semaphores.