Philosophers Project: Step-by-Step Implementation Guide

This guide provides a detailed, step-by-step approach to implementing the Dining Philosophers Problem solution using the odd/even algorithm in C, leveraging pthreads and mutexes.

1. Project Structure

A well-organized project structure is crucial for maintainability and readability. We will organize our project into several files:

- `philo.h` : This header file will contain all necessary structure definitions, function prototypes, and global constants.
- `main.c` : This file will handle argument parsing, initialization of philosophers and forks, thread creation, and simulation management.
- `philosopher.c` : This file will contain the core logic for each philosopher thread, including their routine (thinking, eating, sleeping) and fork acquisition/release.
- `utils.c` : This file will contain utility functions such as time management and logging.
- `Makefile` : To compile the project according to the specified requirements.

Plain Text

```
philo/
├── Makefile
├── philo.h
├── main.c
├── philosopher.c
└── utils.c
```

2. `philo.h` (Header File)

This file will define the essential data structures and function prototypes. It's important to include necessary standard library headers here.

Plain Text

```
#ifndef PHILO_H
# define PHILO_H

# include <pthread.h>
# include <stdio.h>
# include <stdlib.h>
# include <unistd.h>
# include <sys/time.h>

// Define states for logging
# define STATE_THINKING "is thinking"
# define STATE_EATING "is eating"
# define STATE_SLEEPING "is sleeping"
# define STATE_TAKEN_FORK "has taken a fork"
# define STATE_DIED "died"

// Structure to hold program arguments and shared simulation state
typedef struct s_program_args {
    int                 num_philosophers;
    long long           time_to_die;
    long long           time_to_eat;
    long long           time_to_sleep;
    int                 num_times_to_eat; // Optional, 0 if not specified
    long long           start_time;
    pthread_mutex_t     *fork_mutexes; // Array of fork mutexes
    pthread_mutex_t     print_mutex; // Mutex for safe printing
    int                 simulation_should_end; // Flag to stop simulation
    pthread_mutex_t     simulation_end_mutex; // Mutex to protect
simulation_should_end
    int                 finished_eating_count; // Count of philosophers who
finished eating
    pthread_mutex_t     finished_eating_mutex; // Mutex to protect
finished_eating_count
} t_program_args;

// Structure to represent a philosopher
typedef struct s_philosopher {
    int                 id;
    pthread_t           thread;
    int                 left_fork_id;
    int                 right_fork_id;
    long long           last_meal_time;
```

```
    int                 meals_eaten;
    t_program_args      *args; // Pointer to shared program arguments
} t_philosopher;

// Function prototypes
void    *philosopher_routine(void *arg);
long long get_current_time_ms(void);
void    log_state(t_philosopher *philo, const char *state_msg);
int     init_program_args(t_program_args *args, int argc, char **argv);
int     init_philosophers(t_philosopher **philosophers, t_program_args
*args);
void    cleanup(t_philosopher *philosophers, t_program_args *args);

#endif
```

3. `main.c` (Main Program Logic)

This file will contain the `main` function, responsible for parsing command-line arguments, initializing the simulation, creating philosopher threads, and managing the simulation lifecycle.

Plain Text

```
#include "philo.h"

int main(int argc, char **argv) {
    t_program_args  args;
    t_philosopher   *philosophers;
    int             i;

    if (argc < 5 || argc > 6) {
        printf("Usage: %s number_of_philosophers time_to_die time_to_eat
time_to_sleep [number_of_times_each_philosopher_must_eat]\n", argv[0]);
        return (1);
    }

    if (init_program_args(&args, argc, argv) != 0) {
        return (1);
    }

    if (init_philosophers(&philosophers, &args) != 0) {
        cleanup(NULL, &args); // Clean up args if philosopher init fails
        return (1);
    }
```

```
    args.start_time = get_current_time_ms();

    // Create philosopher threads
    i = 0;
    while (i < args.num_philosophers) {
        if (pthread_create(&philosophers[i].thread, NULL,
philosopher_routine, &philosophers[i]) != 0) {
            // Handle error, clean up already created threads/mutexes
            printf("Error creating thread for philosopher %d\n",
philosophers[i].id);
            // TODO: Implement robust cleanup for partially created threads
            cleanup(philosophers, &args);
            return (1);
        }
        i++;
    }

    // Monitor loop (main thread)
    // This loop will check for philosopher death or if all have eaten enough
    while (1) {
        // Check for death
        i = 0;
        while (i < args.num_philosophers) {
            pthread_mutex_lock(&args.simulation_end_mutex);
            if (args.simulation_should_end) {
                pthread_mutex_unlock(&args.simulation_end_mutex);
                break;
            }
            pthread_mutex_unlock(&args.simulation_end_mutex);

            long long current_time = get_current_time_ms();
            if (current_time - philosophers[i].last_meal_time >
args.time_to_die) {
                log_state(&philosophers[i], STATE_DIED);
                pthread_mutex_lock(&args.simulation_end_mutex);
                args.simulation_should_end = 1;
                pthread_mutex_unlock(&args.simulation_end_mutex);
                break;
            }
            i++;
        }
        pthread_mutex_lock(&args.simulation_end_mutex);
        if (args.simulation_should_end) {
            pthread_mutex_unlock(&args.simulation_end_mutex);
            break;
        }
        pthread_mutex_unlock(&args.simulation_end_mutex);
```

```
        // Check if all philosophers have eaten enough (if num_times_to_eat
is set)
        if (args.num_times_to_eat > 0) {
            pthread_mutex_lock(&args.finished_eating_mutex);
            if (args.finished_eating_count == args.num_philosophers) {
                pthread_mutex_unlock(&args.finished_eating_mutex);
                pthread_mutex_lock(&args.simulation_end_mutex);
                args.simulation_should_end = 1;
                pthread_mutex_unlock(&args.simulation_end_mutex);
                break;
            }
            pthread_mutex_unlock(&args.finished_eating_mutex);
        }
        usleep(1000); // Small delay to prevent busy-waiting
    }

    // Join philosopher threads
    i = 0;
    while (i < args.num_philosophers) {
        pthread_join(philosophers[i].thread, NULL);
        i++;
    }

    cleanup(philosophers, &args);

    return (0);
}

// Helper function to initialize program arguments
int init_program_args(t_program_args *args, int argc, char **argv) {
    args->num_philosophers = atoi(argv[1]);
    args->time_to_die = atoll(argv[2]);
    args->time_to_eat = atoll(argv[3]);
    args->time_to_sleep = atoll(argv[4]);
    args->num_times_to_eat = 0;
    if (argc == 6) {
        args->num_times_to_eat = atoi(argv[5]);
    }

    if (args->num_philosophers <= 0 || args->time_to_die <= 0 || \
        args->time_to_eat <= 0 || args->time_to_sleep <= 0 || \
        (argc == 6 && args->num_times_to_eat <= 0)) {
        printf("Invalid arguments. All values must be positive integers.\n");
        return (1);
    }

    args->fork_mutexes = (pthread_mutex_t *)malloc(sizeof(pthread_mutex_t) *
```

```
args->num_philosophers);
    if (!args->fork_mutexes) {
        printf("Error allocating memory for fork mutexes.\n");
        return (1);
    }

    for (int i = 0; i < args->num_philosophers; i++) {
        if (pthread_mutex_init(&args->fork_mutexes[i], NULL) != 0) {
            printf("Error initializing fork mutex %d.\n", i);
            // TODO: Cleanup already initialized mutexes
            free(args->fork_mutexes);
            return (1);
        }
    }

    if (pthread_mutex_init(&args->print_mutex, NULL) != 0) {
        printf("Error initializing print mutex.\n");
        // TODO: Cleanup fork mutexes
        free(args->fork_mutexes);
        return (1);
    }

    if (pthread_mutex_init(&args->simulation_end_mutex, NULL) != 0) {
        printf("Error initializing simulation end mutex.\n");
        // TODO: Cleanup other mutexes
        free(args->fork_mutexes);
        pthread_mutex_destroy(&args->print_mutex);
        return (1);
    }

    if (pthread_mutex_init(&args->finished_eating_mutex, NULL) != 0) {
        printf("Error initializing finished eating mutex.\n");
        // TODO: Cleanup other mutexes
        free(args->fork_mutexes);
        pthread_mutex_destroy(&args->print_mutex);
        pthread_mutex_destroy(&args->simulation_end_mutex);
        return (1);
    }

    args->simulation_should_end = 0;
    args->finished_eating_count = 0;

    return (0);
}

// Helper function to initialize philosophers
int init_philosophers(t_philosopher **philosophers, t_program_args *args) {
    *philosophers = (t_philosopher *)malloc(sizeof(t_philosopher) * args-
```

```
>num_philosophers);
    if (!*philosophers) {
        printf("Error allocating memory for philosophers.\n");
        return (1);
    }

    for (int i = 0; i < args->num_philosophers; i++) {
        (*philosophers)[i].id = i + 1;
        (*philosophers)[i].left_fork_id = i;
        (*philosophers)[i].right_fork_id = (i + 1) % args->num_philosophers;
        (*philosophers)[i].last_meal_time = 0; // Will be set at simulation
start
        (*philosophers)[i].meals_eaten = 0;
        (*philosophers)[i].args = args;
    }
    return (0);
}

// Helper function for cleanup
void cleanup(t_philosopher *philosophers, t_program_args *args) {
    if (philosophers) {
        free(philosophers);
    }
    if (args->fork_mutexes) {
        for (int i = 0; i < args->num_philosophers; i++) {
            pthread_mutex_destroy(&args->fork_mutexes[i]);
        }
        free(args->fork_mutexes);
    }
    pthread_mutex_destroy(&args->print_mutex);
    pthread_mutex_destroy(&args->simulation_end_mutex);
    pthread_mutex_destroy(&args->finished_eating_mutex);
}
```

4. `philosopher.c` (Philosopher Routine)

This file will contain the `philosopher_routine` function, which is the entry point for each philosopher thread. It implements the thinking, eating, and sleeping cycle, incorporating the odd/even algorithm for fork acquisition.

Plain Text

```
#include "philo.h"
```

```c
void *philosopher_routine(void *arg) {
    t_philosopher *philo = (t_philosopher *)arg;
    t_program_args *args = philo->args;

    // Initial last_meal_time for all philosophers
    philo->last_meal_time = args->start_time;

    // Handle single philosopher case
    if (args->num_philosophers == 1) {
        log_state(philo, STATE_TAKEN_FORK);
        usleep(args->time_to_die * 1000); // Wait until death
        // The main thread will detect death and log it
        return (NULL);
    }

    // Philosophers with even IDs start by waiting to avoid immediate
contention
    if (philo->id % 2 == 0) {
        usleep(1000); // Small delay for even philosophers
    }

    while (1) {
        pthread_mutex_lock(&args->simulation_end_mutex);
        if (args->simulation_should_end) {
            pthread_mutex_unlock(&args->simulation_end_mutex);
            break;
        }
        pthread_mutex_unlock(&args->simulation_end_mutex);

        // Thinking
        log_state(philo, STATE_THINKING);

        // Eating
        // Odd/Even algorithm for fork acquisition
        if (philo->id % 2 != 0) { // Odd philosopher
            pthread_mutex_lock(&args->fork_mutexes[philo->left_fork_id]);
            log_state(philo, STATE_TAKEN_FORK);
            pthread_mutex_lock(&args->fork_mutexes[philo->right_fork_id]);
            log_state(philo, STATE_TAKEN_FORK);
        } else { // Even philosopher
            pthread_mutex_lock(&args->fork_mutexes[philo->right_fork_id]);
            log_state(philo, STATE_TAKEN_FORK);
            pthread_mutex_lock(&args->fork_mutexes[philo->left_fork_id]);
            log_state(philo, STATE_TAKEN_FORK);
        }

        log_state(philo, STATE_EATING);
        philo->last_meal_time = get_current_time_ms();
```

```
        usleep(args->time_to_eat * 1000);
        philo->meals_eaten++;

        // Check if philosopher has eaten enough
        if (args->num_times_to_eat > 0 && philo->meals_eaten >= args-
>num_times_to_eat) {
            pthread_mutex_lock(&args->finished_eating_mutex);
            args->finished_eating_count++;
            pthread_mutex_unlock(&args->finished_eating_mutex);
            // No need to break here, main thread will handle simulation end
        }

        // Release forks
        pthread_mutex_unlock(&args->fork_mutexes[philo->left_fork_id]);
        pthread_mutex_unlock(&args->fork_mutexes[philo->right_fork_id]);

        // Sleeping
        log_state(philo, STATE_SLEEPING);
        usleep(args->time_to_sleep * 1000);
    }
    return (NULL);
}
```

5. `utils.c` (Utility Functions)

This file will contain helper functions for time management and logging, ensuring consistent and thread-safe output.

Plain Text

```
#include "philo.h"

// Returns current time in milliseconds
long long get_current_time_ms(void) {
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return (tv.tv_sec * 1000 + tv.tv_usec / 1000);
}

// Logs the state of a philosopher
void log_state(t_philosopher *philo, const char *state_msg) {
    pthread_mutex_lock(&philo->args->print_mutex);
    long long timestamp = get_current_time_ms() - philo->args->start_time;
    printf("%lld %d %s\n", timestamp, philo->id, state_msg);
```

```
    pthread_mutex_unlock(&philo->args->print_mutex);
}
```

6. `Makefile`

The `Makefile` will automate the compilation process, adhering to the project's requirements.

Plain Text

```
NAME = philo

SRCS = main.c \
       philosopher.c \
       utils.c

OBJS = $(SRCS:.c=.o)

CC = cc
CFLAGS = -Wall -Wextra -Werror -pthread

all: $(NAME)

$(NAME): $(OBJS)
    $(CC) $(CFLAGS) $(OBJS) -o $(NAME)

%.o: %.c philo.h
    $(CC) $(CFLAGS) -c $< -o $@

clean:
    rm -f $(OBJS)

fclean:
    rm -f $(OBJS) $(NAME)

re: fclean all

.PHONY: all clean fclean re
```

7. Compilation and Execution

To compile your project, navigate to the `philo` directory in your terminal and run:

Bash

```bash
make
```

This will create an executable named `philo`. You can then run the simulation with the required arguments:

Bash

```bash
./philo number_of_philosophers time_to_die time_to_eat time_to_sleep
[number_of_times_each_philosopher_must_eat]
```

Example:

Bash

```bash
./philo 5 800 200 200
```

This command simulates 5 philosophers, where a philosopher dies if they don't eat within 800ms, eating takes 200ms, and sleeping takes 200ms.

8. Debugging and Testing Strategies

- **Verbose Logging:** Use the `log_state` function extensively to track the exact sequence of events. This is invaluable for identifying race conditions or unexpected behavior.
- **Small Number of Philosophers:** Start with a small number of philosophers (e.g., 2 or 3) to make it easier to trace the execution flow and identify issues.
- **Adjusting Time Parameters:** Experiment with different `time_to_die`, `time_to_eat`, and `time_to_sleep` values. Short `time_to_die` values will quickly reveal starvation issues. Long `time_to_eat` values can increase the likelihood of contention.
- **Memory Leak Detection:** Use tools like `valgrind` to check for memory leaks. This is crucial as the project explicitly states that memory leaks will not be tolerated.

- **Race Condition Detection:** Tools like `helgrind` (part of Valgrind) can help detect threading errors, including race conditions.
- **Edge Cases:** Test with edge cases:
  - `number_of_philosophers = 1`: The single philosopher should die.
  - `number_of_times_each_philosopher_must_eat` set to a small number (e.g., 1 or 5) to ensure the simulation terminates correctly based on meal count.

9. Performance Optimization Tips

While correctness and deadlock prevention are paramount, consider these tips for optimization:

- **Minimize Mutex Locking:** Only lock mutexes when absolutely necessary and release them as soon as possible. Excessive locking can reduce concurrency.
- **Avoid Busy-Waiting:** Do not use `while` loops that constantly check a condition without yielding the CPU (e.g., `while (condition) {}`). Instead, use `usleep` or `pthread_cond_wait` (though `usleep` is generally sufficient for this project's requirements).
- **Efficient Time Management:** `gettimeofday` is generally efficient enough for this project. Avoid more complex timing mechanisms unless profiling reveals a bottleneck.
- **Logging Overhead:** While logging is required, be mindful that excessive `printf` calls can introduce overhead. Ensure the `print_mutex` is used correctly to prevent contention on `stdout`.

By following this comprehensive guide, you will be well-equipped to implement a robust and correct solution to the Dining Philosophers Problem, effectively demonstrating deadlock prevention using the odd/even algorithm.