National University of Computer and Emerging Sciences

Dining Philosophers Problem Using System Call

Group Members

Samra Shahid……22K-4585
Laiba Khan……22K-4610

Supervised by

Miss Anaum Hamid

FAST School of Computing

National University of Computer and Emerging Sciences

Karachi, Pakistan

May 2024

Dining Philosopher Problem Using System Call

1. Introduction

The Dining Philosophers problem is a classic synchronization problem in computer science that illustrates the challenges of resource allocation and deadlock avoidance in concurrent systems. The problem involves a group of philosophers sitting around a dining table, each philosopher alternating between thinking and eating. The philosophers share a finite number of chopsticks placed between them, and to eat, a philosopher must acquire both the chopsticks adjacent to them.

The challenge arises from the potential for deadlock, where each philosopher holds one chopstick and waits indefinitely for the other. Various solutions exist to address this issue, including the use of semaphores, mutexes, or monitors to control access to the shared chopsticks.

In this project, we aim to explore an alternative approach to solving the Dining Philosophers problem using system calls. System calls provide a mechanism for processes to interact with the operating system kernel, allowing us to implement synchronization and resource management at the kernel level. By leveraging system calls, we can design a solution that ensures fair access to chopsticks while avoiding deadlock. We started working on our project by the end of April and concluded it on 12th May 2024.

2. Features

Our program has the following key features:

1. **Concurrency Management:** Semaphores are used to control access to shared resources, in this case, the chopsticks. Each chopstick is represented by a semaphore. Philosophers must acquire the semaphore (chopstick) before eating and release it afterward.
2. **Deadlock Avoidance:** By carefully controlling the acquisition and release of semaphores, deadlock can be avoided. Deadlock occurs when each philosopher picks up one chopstick and waits indefinitely for the other, leading to a circular dependency. Semaphore-based solutions aim to prevent this scenario.
3. **Mutual Exclusion:** Semaphores ensure that only one philosopher can access a chopstick at a time. When a philosopher wants to eat, they must acquire both adjacent chopsticks. Semaphores enforce mutual exclusion, ensuring that only one philosopher can hold a chopstick at any given time.
4. **Resource Sharing:** The chopsticks are shared resources among the philosophers. Semaphores allow for controlled sharing of these resources by enforcing access policies. Philosophers must wait if a chopstick is currently in use by another philosopher.
5. **Synchronization:** Semaphores are used to synchronize the actions of the philosophers. They coordinate the acquisition and release of chopsticks to ensure that philosophers do not interfere with each other's eating process.
6. **Implementation Flexibility:** Semaphore-based solutions offer flexibility in implementation. Different semaphore initialization values and algorithms can be employed to achieve specific synchronization goals, such as fairness or efficiency.

7. **Scalability:** Semaphore-based solutions can scale to accommodate varying numbers of philosophers and chopsticks. The use of semaphores allows for the coordination of a large number of concurrent processes, making them suitable for real-world applications with multiple resource-sharing entities.

2. Technology Used

Programming Language: C language
VMware Work Station 16
Platform: Ubuntu 16.04

3. Code Snippets

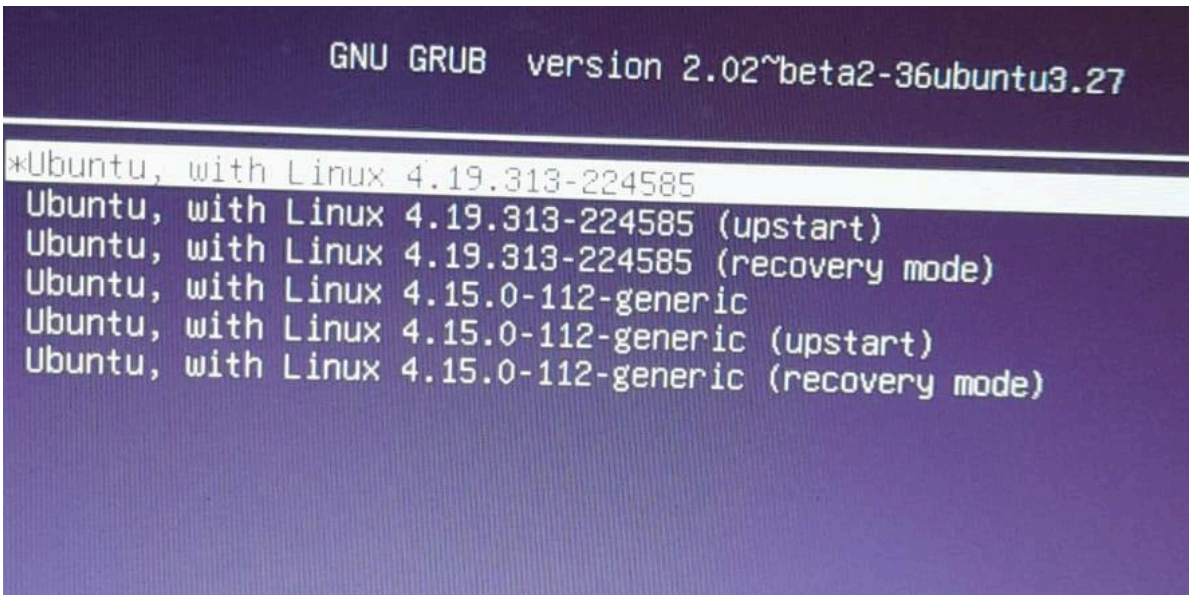
.C file for the dining philosopher problem

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <semaphore.h>
#include <sys/types.h>
#include <sys/syscall.h>

#define N 5 // Number of philosophers

sem_t chopstick[N];

void *eat_think_repeat(void *arg) {
   int philosopher_id = *((int *)arg);
   int left_chopstick = philosopher_id;
   int right_chopstick = (philosopher_id + 1) % N;

   while (1) {
      // Thinking
      printf("Philosopher %d is thinking\n", philosopher_id);
      sleep(rand() % 3 + 1);

      // Pick up chopsticks
      sem_wait(&chopstick[left_chopstick]);
      printf("Philosopher %d picked up left chopstick %d\n", philosopher_id, left_chopstick);
      sem_wait(&chopstick[right_chopstick]);
      printf("Philosopher %d picked up right chopstick %d\n", philosopher_id,
right_chopstick);

      // Eating
      printf("Philosopher %d is eating\n", philosopher_id);
      sleep(rand() % 3 + 1);

      // Put down chopsticks
      sem_post(&chopstick[left_chopstick]);
```

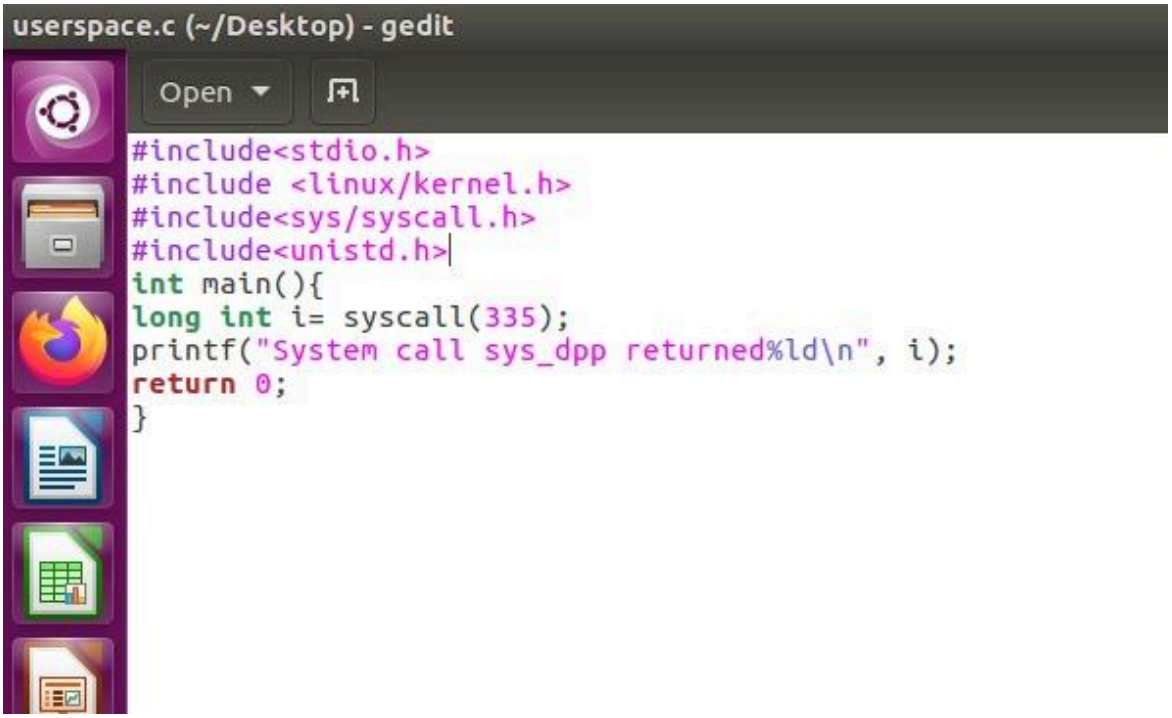
```
        printf("Philosopher %d put down left chopstick %d\n", philosopher_id, left_chopstick);
        sem_post(&chopstick[right_chopstick]);
        printf("Philosopher %d put down right chopstick %d\n", philosopher_id,
right_chopstick);
    }

    return NULL;
}

int main() {
    syscall(333);
    int i;
    int philosopher_id[N];

    // Initialize semaphores for chopsticks
    for (i = 0; i < N; i++) {
        sem_init(&chopstick[i], 0, 1);
    }

    // Create philosopher threads
    pthread_t philosopher[N];
    for (i = 0; i < N; i++) {
        philosopher_id[i] = i;
        pthread_create(&philosopher[i], NULL, eat_think_repeat, &philosopher_id[i]);
    }

    // Join philosopher threads
    for (i = 0; i < N; i++) {
        pthread_join(philosopher[i], NULL);
    }

    // Destroy semaphores
    for (i = 0; i < N; i++) {
        sem_destroy(&chopstick[i]);
    }

    return 0;
}
```

Kernel Modification

1. Kernel configured using roll number : 22K-4585



2. Kernel Level Code



3. Successful Execution

```
osproject@osproject-virtual-machine:~$ cd Desktop
osproject@osproject-virtual-machine:~/Desktop$ gedit userspace.c
osproject@osproject-virtual-machine:~/Desktop$ gcc userspace.c
osproject@osproject-virtual-machine:~/Desktop$ ./a.out
System call sys_dpp returned0
osproject@osproject-virtual-machine:~/Desktop$ dmesg
[    0.000000] Linux version 4.19.313-224585 (root@osproject-virtua
cc version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.12)) #1 SMP
:29:18 PKT 2024
[    0.000000] Command line: BOOT_IMAGE=/boot/vmlinuz-4.19.313-2245
```

```
osproject@osproject-virtual-machine: ~/Desktop
[ 1765.945870] Samra Shahid and Laiba Khan
[ 1765.945878] Dining Philosopher Problem
[ 1981.374613] Samra Shahid and Laiba Khan
[ 1981.374622] Dining Philosopher Problem
osproject@osproject-virtual-machine:~/Desktop$
osproject@osproject-virtual-machine:~/Desktop$
osproject@osproject-virtual-machine:~/Desktop$
osproject@osproject-virtual-machine:~/Desktop$
osproject@osproject-virtual-machine:~/Desktop$
osproject@osproject-virtual-machine:~/Desktop$
osproject@osproject-virtual-machine:~/Desktop$
osproject@osproject-virtual-machine:~/Desktop$ gedit dining.c
osproject@osproject-virtual-machine:~/Desktop$ gcc -pthread dining.c -o dining
osproject@osproject-virtual-machine:~/Desktop$ ./dining
Philosopher 2 is thinking
Philosopher 4 is thinking
Philosopher 3 is thinking
Philosopher 0 is thinking
Philosopher 1 is thinking
Philosopher 3 picked up left chopstick 3
Philosopher 3 picked up right chopstick 4
Philosopher 3 is eating
^C
```

```
f_event_max_sample_rate to 15250
[ 1474.422162] perf: interrupt took too long (16279 > 1622
f_event_max_sample_rate to 12250
[ 1610.915556] INFO: NMI handler (perf_event_nmi_handler) t
1.677 msecs
[ 1610.915574] perf: interrupt took too long (21637 > 2034
f_event_max_sample_rate to 9000
[ 1765.945870] Samra Shahid and Laiba Khan
[ 1765.945878] Dining Philosopher Problem
[ 1981.374613] Samra Shahid and Laiba Khan
[ 1981.374622] Dining Philosopher Problem
[ 2080.232816] Samra Shahid and Laiba Khan
[ 2080.232826] Dining Philosopher Problem
osproject@osproject-virtual-machine:~/Desktop$
```

4. Conclusion

In this project, we embarked on the task of solving the Dining Philosophers problem using system calls, with the goal of exploring alternative synchronization mechanisms at the kernel level. While the project encountered limitations in implementing system calls for managing

chopsticks directly, significant insights were gained into the challenges and opportunities of kernel-level synchronization.

The primary achievement of the project was the successful implementation of a system call for a printk statement, demonstrating the basic framework for interacting with the kernel from user space. However, due to the complexity and constraints of kernel programming, extending this to encompass the management of chopsticks proved to be challenging.

As a result, the dining code itself, including the synchronization logic and resource management, was executed at the user level. Despite this limitation, valuable lessons were learned about the design and implementation of system calls, as well as the intricacies of coordinating concurrent processes within the kernel environment.

5. **Github Repository**

https://github.com/laibak24/Dining-Philosopher-Problem