Name:Arunima Chaudhuri
Roll No: 23CSM1R04

Assignment 4.1 Four Classical IPC problems using system V IPC

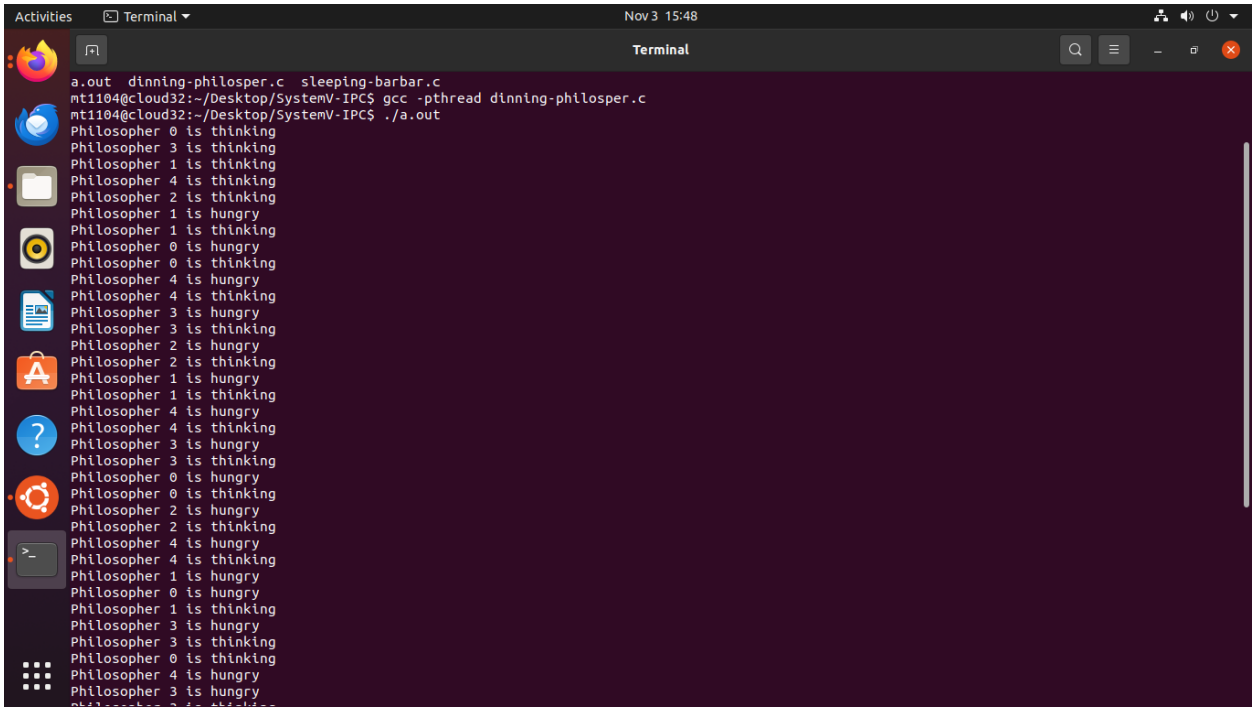
Dining Philosopher

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define NUM_PHILOSOPHERS 5
#define NUM_FORKS 5

#define LEFT (i)
#define RIGHT ((i+1)%NUM_PHILOSOPHERS)

#define THINKING 0
#define HUNGRY 1
#define EATING 2

int state[NUM_PHILOSOPHERS];
int phil[NUM_PHILOSOPHERS] = {0, 1, 2, 3, 4};
int forks[NUM_FORKS];

int semid;

union semun {
    int val;
    struct semid_ds *buf;
    unsigned short *array;
};

void grab_forks(int i) {
    struct sembuf semaphore_operation[2];

    semaphore_operation[0].sem_num = LEFT;
    semaphore_operation[0].sem_op = -1;
```

```
    semaphore_operation[0].sem_flg = 0;

    semaphore_operation[1].sem_num = RIGHT + NUM_PHILOSOPHERS;
    semaphore_operation[1].sem_op = -1;
    semaphore_operation[1].sem_flg = 0;

    if (semop(semid, semaphore_operation, 2) == -1) {
        perror("semop");
        exit(EXIT_FAILURE);
    }
}

void put_away_forks(int i) {
    struct sembuf semaphore_operation[2];

    semaphore_operation[0].sem_num = LEFT;
    semaphore_operation[0].sem_op = 1;
    semaphore_operation[0].sem_flg = 0;

    semaphore_operation[1].sem_num = RIGHT + NUM_PHILOSOPHERS;
    semaphore_operation[1].sem_op = 1;
    semaphore_operation[1].sem_flg = 0;

    if (semop(semid, semaphore_operation, 2) == -1) {
        perror("semop");
        exit(EXIT_FAILURE);
    }
}

void test(int i) {
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] !=
EATING) {
        state[i] = EATING;
        printf("Philosopher %d is eating\n", i);
    }
}

void think(int i) {
    printf("Philosopher %d is thinking\n", i);
    sleep(rand() % 3 + 1); // Simulate thinking
}

void eat(int i) {
```

```
    printf("Philosopher %d is hungry\n", i);
    grab_forks(i);
    test(i);
    put_away_forks(i);
}

void *philosopher(void *arg) {
    int i = *((int*)arg);

    while (1) {
        think(i);
        eat(i);
    }
}

int main() {
    int i;

    // Initialize semaphores
    semid = semget(IPC_PRIVATE, 2 * NUM_PHILOSOPHERS, 0666 | IPC_CREAT);
    if (semid == -1) {
        perror("semget");
        exit(EXIT_FAILURE);
    }

    union semun arg;
    arg.val = 1;

    for (i = 0; i < NUM_PHILOSOPHERS; i++) {
        if (semctl(semid, i, SETVAL, arg) == -1) {
            perror("semctl");
            exit(EXIT_FAILURE);
        }
        if (semctl(semid, i + NUM_PHILOSOPHERS, SETVAL, arg) == -1) {
            perror("semctl");
            exit(EXIT_FAILURE);
        }
    }

    // Initialize forks
    for (i = 0; i < NUM_FORKS; i++) {
        forks[i] = 1;
    }
```

```
    // Create philosopher threads
    pthread_t philosophers[NUM_PHILOSOPHERS];

    for (i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_create(&philosophers[i], NULL, philosopher, (void
*)&phil[i]);
    }

    // Wait for threads to finish (this won't happen)
    for (i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_join(philosophers[i], NULL);
    }

    // Clean up semaphores
    semctl(semid, 0, IPC_RMID);

    return 0;
}
```



Sleeping Barber

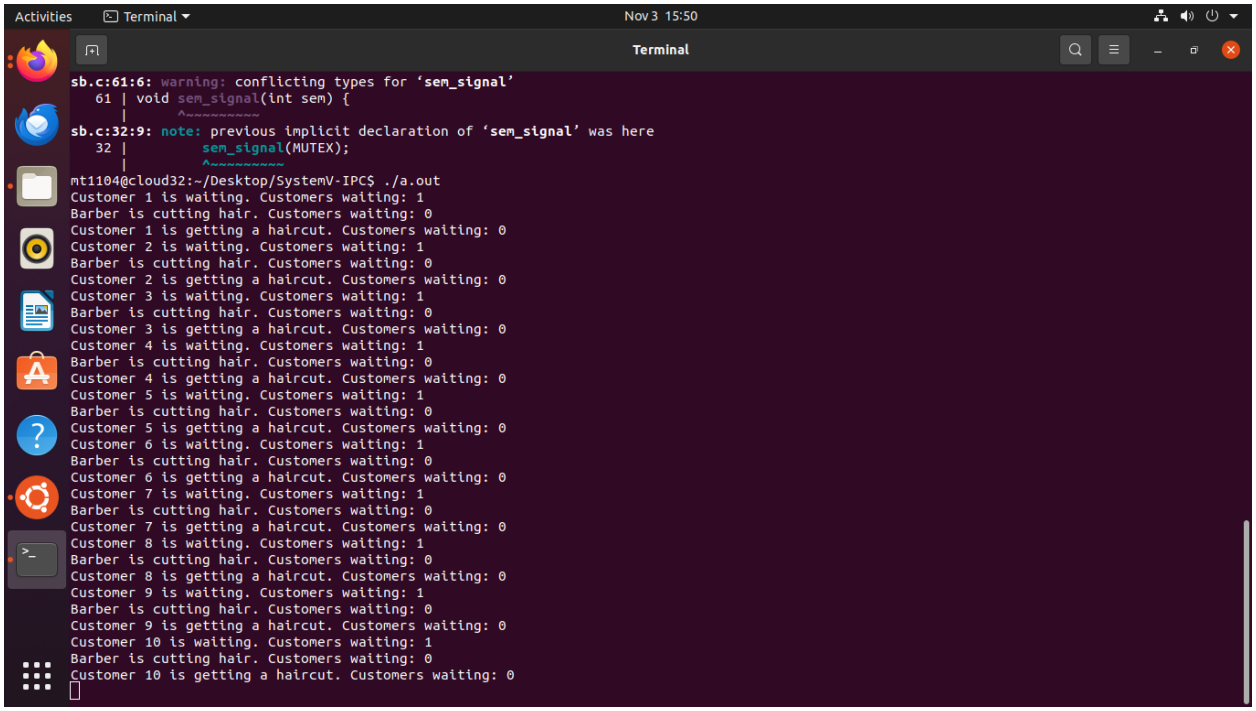
```
#include <stdio.h>
```

```c
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <string.h>
#include <pthread.h>


#define KEY 1234
#define WAITING_ROOM_SIZE 5

// Define semaphore union
union semun {
      int val;
      struct semid_ds *buf;
      unsigned short *array;
      struct seminfo *__buf;
};

// Function to create and initialize semaphores
int create_semaphore_set(int n, int initial_value) {
      int semid = semget(KEY, n, IPC_CREAT | 0666);
      if (semid == -1) {
      perror("semget");
      exit(EXIT_FAILURE);
      }

      union semun arg;
      arg.val = initial_value;

      for (int i = 0; i < n; i++) {
      if (semctl(semid, i, SETVAL, arg) == -1) {
            perror("semctl");
            exit(EXIT_FAILURE);
      }
      }

      return semid;
}

// Function to create and initialize shared memory
```

```c
int create_shared_memory(size_t size) {
      int shmid = shmget(KEY, size, IPC_CREAT | 0666);
      if (shmid == -1) {
      perror("shmget");
      exit(EXIT_FAILURE);
      }

      return shmid;
}

int main() {
      // Create semaphores
      int barber_chair = create_semaphore_set(1, 1);
      int customers = create_semaphore_set(1, 0);
      int barber_sleeping = create_semaphore_set(1, 0);
      int barber_awake = create_semaphore_set(1, 0);

      // Create shared memory for waiting room
      int waiting_room = create_shared_memory(WAITING_ROOM_SIZE *
sizeof(int));

      pid_t barber_pid = fork();

      if (barber_pid == -1) {
      perror("fork");
      exit(EXIT_FAILURE);
      }

      if (barber_pid == 0) {
      // Barber process
      while (1) {
            sem_wait(customers);
            sem_wait(barber_chair);

            // Read customer ID from waiting room
            int *waiting_room_ptr = shmat(waiting_room, NULL, 0);
            int customer_id = waiting_room_ptr[0];
            shmdt(waiting_room_ptr);

            printf("Barber is cutting hair of customer %d\n", customer_id);
            sleep((rand() % 3) + 1);

            printf("Barber finished cutting hair of customer %d\n",
```

```
customer_id);
            sem_post(barber_awake);
      }
      } else {
      // Customer processes
      for (int i = 0; i < 10; i++) {
            pid_t customer_pid = fork();

            if (customer_pid == -1) {
                  perror("fork");
                  exit(EXIT_FAILURE);
            }

            if (customer_pid == 0) {
                  // Customer process
                  sem_wait(barber_chair);
                  sem_post(customers);

                  // Find an empty seat in waiting room
                  int *waiting_room_ptr = shmat(waiting_room, NULL, 0);
                  for (int j = 0; j < WAITING_ROOM_SIZE; j++) {
                  if (waiting_room_ptr[j] == 0) {
                        waiting_room_ptr[j] = i;
                        printf("Customer %d is waiting in the waiting
room.\n", i);
                        break;
                  }
                  }
                  shmdt(waiting_room_ptr);

                  sem_wait(barber_awake);
                  printf("Customer %d is having a haircut.\n", i);
                  exit(EXIT_SUCCESS);
            }
      }

      // Parent process (main)
      for (int i = 0; i < 10; i++) {
            wait(NULL);
      }

      // Clean up semaphores and shared memory
      semctl(barber_chair, 0, IPC_RMID);
```

```
        semctl(customers, 0, IPC_RMID);
        semctl(barber_sleeping, 0, IPC_RMID);
        semctl(barber_awake, 0, IPC_RMID);
        shmctl(waiting_room, IPC_RMID, NULL);
        }

        return 0;
}
```



Reader Writer

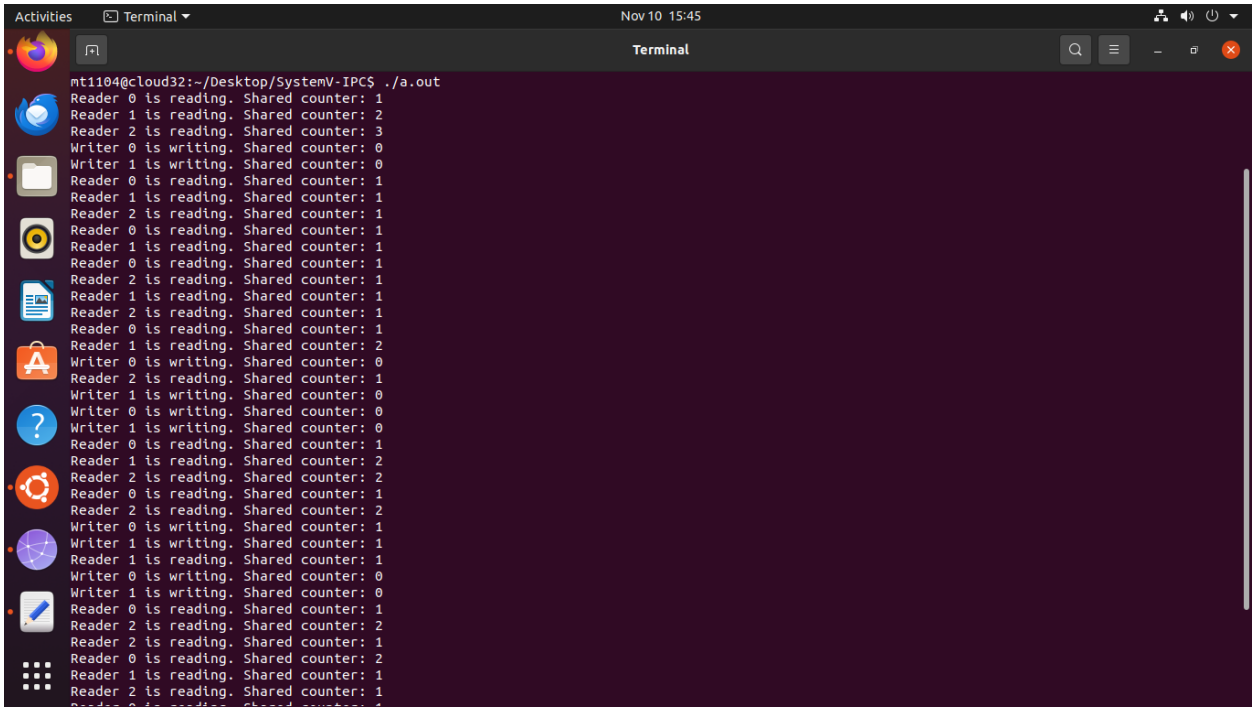
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/wait.h>
#include <sys/shm.h>
#define SHM_KEY ftok(".", 'R')
#define SEM_KEY ftok(".", 'S')
#define NUM_READERS 3
#define NUM_WRITERS 2
int shmid;
```

```
int semid;
void reader(int id);
void writer(int id);
void initialize();
void destroy();
union semun {
int val;
struct semid_ds *buf;
unsigned short *array;
};
int main() {
int i;
pid_t pid;
// Create and initialize shared memory and semaphores
initialize();
// Create reader processes
for (i = 0; i < NUM_READERS; i++) {
if ((pid = fork()) == 0) {
reader(i);
exit(0);
}
}
// Create writer processes
for (i = 0; i < NUM_WRITERS; i++) {
if ((pid = fork()) == 0) {
writer(i);
exit(0);
}
}
// Wait for all child processes to finish
for (i = 0; i < NUM_READERS + NUM_WRITERS; i++) {

wait(NULL);
}
// Cleanup and destroy shared memory and semaphores
destroy();
return 0;
}
void initialize() {
// Create shared memory segment
if ((shmid = shmget(SHM_KEY, sizeof(int), IPC_CREAT | 0666)) == -1) {
perror("shmget");
exit(1);
```

```
}
// Attach shared memory
int *shared_counter = shmat(shmid, NULL, 0);
*shared_counter = 0;
// Create semaphore set
if ((semid = semget(SEM_KEY, 3, IPC_CREAT | 0666)) == -1) {
perror("semget");
exit(1);
}
// Initialize semaphores
union semun arg;
arg.val = 1; // semaphore for controlling access to shared_counter
if (semctl(semid, 0, SETVAL, arg) == -1) {
perror("semctl");
exit(1);
}
arg.val = 0; // semaphore for counting the number of readers
if (semctl(semid, 1, SETVAL, arg) == -1) {
perror("semctl");
exit(1);
}
arg.val = 1; // semaphore for controlling access to shared resource
if (semctl(semid, 2, SETVAL, arg) == -1) {
perror("semctl");
exit(1);
}
}
void destroy() {
// Detach shared memory
int *shared_counter = shmat(shmid, NULL, 0);
shmdt(shared_counter);
// Remove shared memory segment
if (shmctl(shmid, IPC_RMID, NULL) == -1) {
perror("shmctl");
exit(1);
}
// Remove semaphore set
if (semctl(semid, 0, IPC_RMID) == -1) {
perror("semctl");
exit(1);
}
}
void reader(int id) {
```

```
int *shared_counter = shmat(shmid, NULL, 0);
while (1) {
// Wait for access to shared_counter
struct sembuf wait_sem = {0, -1, 0};
semop(semid, &wait_sem, 1);
// Increment the number of readers
(*shared_counter)++;
// If it's the first reader, lock the resource for writers
if (*shared_counter == 1) {
struct sembuf lock_resource_sem = {2, -1, 0};
semop(semid, &lock_resource_sem, 1);
}
// Release access to shared_counter
struct sembuf signal_sem = {0, 1, 0};
semop(semid, &signal_sem, 1);
// Read from the shared resource
printf("Reader %d is reading. Shared counter: %d\n", id, *shared_counter);
// Wait for access to shared_counter
semop(semid, &wait_sem, 1);
// Decrement the number of readers
(*shared_counter)--;
// If it's the last reader, release the resource for writers
if (*shared_counter == 0) {
struct sembuf release_resource_sem = {2, 1, 0};
semop(semid, &release_resource_sem, 1);
}
// Release access to shared_counter
semop(semid, &signal_sem, 1);
// Sleep for a random period before reading again
sleep(rand() % 3);
}
}
void writer(int id) {
int *shared_counter = shmat(shmid, NULL, 0);
while (1) {
// Wait for access to the resource for writers
struct sembuf wait_resource_sem = {2, -1, 0};
semop(semid, &wait_resource_sem, 1);
// Write to the shared resource
printf("Writer %d is writing. Shared counter: %d\n", id, *shared_counter);
// Release access to the resource for writers
struct sembuf signal_resource_sem = {2, 1, 0};
semop(semid, &signal_resource_sem, 1);
```

```
// Sleep for a random period before writing again
sleep(rand()%5);
}
}
```



Producer Consumer

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#define SHARED_MEMORY_KEY 1234
#define SEMAPHORE_KEY 5678
#define BUFFER_SIZE 5
// Structure of the shared circular buffer
struct CircularBuffer {
int buffer[BUFFER_SIZE];
int in;
int out;
};
```

```
void producer(int shmid, int sem_id, int totalItems) {
struct CircularBuffer *buffer = (struct CircularBuffer *)shmat(shmid, 0,
0);
int itemsProduced = 0;
while (itemsProduced < totalItems) {
struct sembuf wait_produce[1] = {{0, -1, SEM_UNDO}}; // Wait for space in
the buffer
semop(sem_id, wait_produce, 1);
int item = rand() % 100; // Generate random item
printf("Producing item: %d\n", item);
buffer->buffer[buffer->in] = item;
buffer->in = (buffer->in + 1) % BUFFER_SIZE;
struct sembuf signal_produce[1] = {{1, 1, SEM_UNDO}}; // Signal that an
item has been produced
semop(sem_id, signal_produce, 1);
itemsProduced++;
}
}

void consumer(int shmid, int sem_id, int totalItems) {
struct CircularBuffer *buffer = (struct CircularBuffer *)shmat(shmid, 0,
0);
int itemsConsumed = 0;
while (itemsConsumed < totalItems) {
struct sembuf wait_consume[1] = {{1, -1, SEM_UNDO}}; // Wait for an item in
the buffer
semop(sem_id, wait_consume, 1);
int item = buffer->buffer[buffer->out];
printf("Consuming item: %d\n", item);
buffer->out = (buffer->out + 1) % BUFFER_SIZE;
struct sembuf signal_consume[1] = {{0, 1, SEM_UNDO}}; // Signal that an
item has been consumed
semop(sem_id, signal_consume, 1);
itemsConsumed++;
}
}

int main() {
int shmid = shmget(SHARED_MEMORY_KEY, sizeof(struct CircularBuffer), 0666 |
IPC_CREAT);
if (shmid == -1) {
perror("Shared memory creation failed");
exit(EXIT_FAILURE);
```

```
}
int sem_id = semget(SEMAPHORE_KEY, 2, 0666 | IPC_CREAT);
if (sem_id == -1) {perror("Semaphore creation failed");
exit(EXIT_FAILURE);
}
structCircularBuffer *buffer = (struct CircularBuffer *)shmat(shmid, 0, 0);
buffer->in = 0;
buffer->out = 0;
int totalItems = 10; // Define the total number of items to be produced and
consumed
semctl(sem_id, 0, SETVAL, BUFFER_SIZE); // Initialize semaphore 0 to buffer
size (indicating
available space)
semctl(sem_id, 1, SETVAL, 0); // Initialize semaphore 1 to 0 (indicating no
items initially)
int pid = fork();
if (pid == 0) {
producer(shmid, sem_id, totalItems);
} else if (pid > 0) {
consumer(shmid, sem_id, totalItems);
} else {
perror("Fork failed");
exit(EXIT_FAILURE);}

shmdt(buffer);
// Remove shared memory and semaphore
shmctl(shmid, IPC_RMID, NULL);
semctl(sem_id, 0, IPC_RMID);

return 0;
}
```

```
Producing item: 2
Consuming item: 2
Producing item: 7
Consuming item: 7
Producing item: 1
Consuming item: 1
Producing item: 4
Consuming item: 4
Producing item: 9
Consuming item: 9
Producing item: 6
Consuming item: 6
Producing item: 5
Consuming item: 5
Producing item: 3
Consuming item: 3
Producing item: 8
Consuming item: 8
```