OS Project

Ice-Cream Factory

Members:

Abdul Kabeer 23K-3004

Jahanzaib Hussain Mirza 23K-3011

Maaz Khan 23K-3036

Qasim Ali 23K-3002

Introduction:

This project implements an inter-process communication (IPC) based Ice Cream Factory system using **POSIX shared memory**, **semaphores**, and **mutex synchronization** in the C programming language. The purpose of this system is to simulate real-world order processing in a multi-step ice cream production pipeline, involving **customer order placement**, **machine processing**, and **synchronized coordination** between multiple processes and threads. Shared memory is used as a centralized communication buffer, while semaphores and mutexes control access to the buffer to prevent race conditions and ensure data consistency.

The setup initializes a shared memory region that holds a circular buffer of orders, along with metadata and named semaphores for producer-consumer coordination. Customer processes interact with this setup by placing orders in the buffer, waiting for processing, and receiving a signal upon completion.

Core Functionalities:

1. Shared Memory Setup

- A shared memory segment (/icecream_shm) is used to hold the order buffer and metadata.
- mmap() maps the shared memory for access across processes.

2. Producer Process

- A single producer thread continuously pulls orders from shared memory (customer orders) and pushes them to the first machine's buffer (Mixing).

3. Multi-Step Processing Pipeline

- There are three stages (machines): **Mixing**, **Freezing**, and **Packaging**.
- Each machine has its own circular buffer to handle orders.
- Each machine stage uses multiple threads (3 per machine) to parallelize order processing.

4. Inter-Process Notification

- Once the order is fully packaged, a signal (SIGUSR1) is sent to the original customer's process to indicate completion.

Dry run:

initializer.c:

Line of Code	Explanation	Updated Variables / Values
#include<stdio.h>	Includes standard I/O functions.	-
#include<stdlib.h>	Includes standard library functions (e.g., exit).	-
#include<unistd.h>	Includes POSIX API for system calls like mmap.	-
#include<sys/mman.h>	Provides memory mapping declarations.	-
#include<string.h>	Includes string handling functions.	-
#include<fcntl.h>	Includes file control options (e.g., O_CREAT).	-
#include<semaphore.h>	Includes POSIX semaphore APIs.	-
#define SHM_NAME "/icecream_shm"	Defines shared memory name.	SHM_NAME = "/icecream_shm"
#define BUFFER_SIZE 10	Defines the size of the order buffer.	BUFFER_SIZE = 10

typedef enum { ... } Flavor;	Defines 8 ice cream flavors as an enum.	-
typedef struct { ... } Order;	Defines the structure for each ice cream order.	-
typedef struct { ... } SharedMemory;	Defines the structure stored in shared memory.	-
int shm_fd = shm_open(SHM_NAME, O_CREAT | O_RDWR, 0666);	Creates or opens a shared memory object for reading and writing.	shm_fd = some valid file descriptor
if (shm_fd == -1) { perror("shm_open"); exit(1); }	Checks if shared memory creation failed; exits on failure.	-
ftruncate(shm_fd, sizeof(SharedMemory));	Sets the size of the shared memory object to hold SharedMemory.	-
SharedMemory* shm_ptr = mmap(...);	Maps the shared memory into the process address space.	shm_ptr -> valid mapped address
shm_ptr->order_in = 0;	Initializes circular buffer insert index.	order_in = 0
shm_ptr->order_out = 0;	Initializes circular buffer remove index.	order_out = 0
shm_ptr->count = 0;	Initializes order count to zero.	count = 0
pthread_mutex_init(&shm_ptr->order_mutex, NULL);	Initializes a mutex for thread-safe access to shared memory.	order_mutex -> initialized
const char* sem_name_in = "/order_semaphore_in";	Defines name for the "full" semaphore.	sem_name_in = "/order_semaphore_in"
const char* sem_name_out = "/order_semaphore_out";	Defines name for the "empty" semaphore.	sem_name_out = "/order_semaphore_out"

sem_t* sem1 = sem_open(..., 0);	Creates named "full" semaphore with initial value 0.	sem1 -> valid semaphore pointer
sem_t* sem2 = sem_open(..., BUFFER_SIZE);	Creates named "empty" semaphore with initial value 10.	sem2 -> valid semaphore pointer
if (sem1 == SEM_FAILED) { perror(...); exit(EXIT_FAILURE); }	Checks if full semaphore creation failed.	-
if (sem2 == SEM_FAILED) { perror(...); exit(EXIT_FAILURE); }	Checks if empty semaphore creation failed.	-
strncpy(shm_ptr->sem_name_full, sem_name_in, ...);	Copies full semaphore name into shared memory struct.	sem_name_full = "/order_semaphore_in"
shm_ptr->sem_name_full[...] = '\0';	Ensures string is null-terminated.	-
strncpy(shm_ptr->sem_name_empty, sem_name_out, ...);	Copies empty semaphore name into shared memory struct.	sem_name_empty = "/order_semaphore_out"
shm_ptr->sem_name_empty[...] = '\0';	Ensures string is null-terminated.	-
printf("Semaphores created and initialized.\n");	Prints confirmation message.	-

customer_order_place.c:

Line of Code	**Explanation**	**Updated Variables / Values**
#include<stdio.h>	**Standard input/output library.**	**-**

Line of Code	Explanation	Updated Variables / Values
#include<stdlib.h>	Standard library for functions like exit().	-
#include<unistd.h>	Provides access to POSIX APIs (e.g., getpid(), sleep()).	-
#include<sys/mman.h>	For shared memory mapping (mmap).	-
#include<string.h>	For string operations like strncpy.	-
#include<fcntl.h>	File control options (O_CREAT, etc.).	-
#include<semaphore.h>	For POSIX semaphores.	-
#include<signal.h>	For signal handling (SIGUSR1).	-
#include<errno.h>	For error codes (e.g., EAGAIN).	-
#include<pthread.h>	For pthread mutex.	-
#define SHM_NAME "/icecream_shm"	Defines shared memory object name.	SHM_NAME = "/icecream_shm"
#define BUFFER_SIZE 10	Max orders in the buffer.	BUFFER_SIZE = 10
typedef enum {...}	Enum defining 8 ice cream flavors.	Flavor enum declared.
typedef struct { ... } Order;	Structure to represent a customer's ice cream order.	Order struct defined.

Line of Code	Explanation	Updated Variables / Values
typedef struct { ... } SharedMemory;	Shared memory layout containing the order buffer and control info.	SharedMemory struct defined.
const char* getFlavorName(...)	Returns string name of a Flavor enum value.	Helper function available.
void handle_sigusr1(int sig)	Signal handler for order completion. Prints confirmation.	-
signal(SIGUSR1, handle_sigusr1);	Registers signal handler for SIGUSR1.	Handler registered for current process.
int shm_fd = shm_open(...);	Opens shared memory created by initializer.	shm_fd = valid descriptor or error.
if (shm_fd == -1) {...}	Checks for shared memory open failure.	-
SharedMemory* shm_ptr = mmap(...);	Maps shared memory to process space.	shm_ptr -> valid address
sem_t* sem_full = sem_open(...);	Opens existing "full" semaphore (order count).	sem_full -> valid handle or error
sem_t* sem_empty = sem_open(...);	Opens existing "empty" semaphore (free slots).	sem_empty -> valid handle or error
`if (sem_full == SEM_FAILED		...)`
Order order;	Declares an Order object to store user input.	-
fgets(order.custname, ...)	Reads customer name from stdin.	e.g., custname = "Alice"

Line of Code	Explanation	Updated Variables / Values
order.custname[strcspn(...)] = '\0';	Removes trailing newline from input.	custname is null-terminated.
scanf("%d", &flavor_choice);	Reads flavor selection from user.	e.g., flavor_choice = 2
order.flavor = (Flavor)flavor_choice;	Converts int to Flavor enum.	order.flavor = STRAWBERRY
scanf("%d", &order.quantity);	Reads quantity.	e.g., quantity = 3
order.customer_pid = getpid();	Saves current process PID.	e.g., customer_pid = 4321
if (sem_trywait(sem_empty) == -1) {...}	Attempts to decrement empty slots.	Waits if no space (non-blocking first).
sem_wait(sem_empty);	Blocks until space becomes available.	Proceed only when a slot is free.
pthread_mutex_lock(&shm_ptr->order_mutex);	Locks mutex to safely write order.	Critical section begins.
shm_ptr->count++;	Increments total order count.	e.g., count = 5
order.orderid = shm_ptr->count;	Assigns order ID.	orderid = 5
shm_ptr->iceCreamOrders[shm_ptr->order_in] = order;	Inserts order at order_in index.	Buffer updated.
shm_ptr->order_in = (shm_ptr->order_in + 1) % BUFFER_SIZE;	Circular increment of order_in.	e.g., order_in = 6
pthread_mutex_unlock(&shm_ptr->order_mutex);	Releases mutex.	Critical section ends.

Line of Code	Explanation	Updated Variables / Values
sleep(1);	**Waits 1 second (simulate processing delay).**	**-**
sem_post(sem_full);	**Increments the full count (new order available).**	**sem_full++**
pause();	**Waits for signal (SIGUSR1).**	**Blocks until notified.**

ice_cream_processor.c:

Line of Code	Explanation	Updated Variables
int shm_fd = shm_open(SHM_NAME, O_RDWR, 0666);	Open shared memory	shm_fd
if (shm_fd == -1) { perror("shm_open"); exit(EXIT_FAILURE); }	Error check for shared memory open	
shm_ptr = mmap(NULL, sizeof(SharedMemory), PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);	Map shared memory	shm_ptr
sem_full = sem_open(shm_ptr->sem_name_full, 0);	Open full semaphore using name in shared memory	sem_full
sem_empty = sem_open(shm_ptr->sem_name_empty, 0);	Open empty semaphore using name in shared memory	sem_empty
init_machine_buffer(&machine1);	Initialize buffer for machine 1	machine1
init_machine_buffer(&machine2);	Initialize buffer for machine 2	machine2
init_machine_buffer(&machine3);	Initialize buffer for machine 3	machine3
pthread_create(&producer, NULL, producer_thread, NULL);	Create thread for producer	producer
pthread_create(&mix_threads[i], ..., &args1);	Create mixing machine threads	mix_threads
pthread_create(&freeze_threads[i], ..., &args2);	Create freezing machine threads	freeze_threads
pthread_create(&pack_threads[i], ..., &args3);	Create packaging machine threads	pack_threads
pthread_join(producer, NULL);	Wait for producer thread	

sem_wait(sem_full);	Wait until there is at least one full slot in shared buffer	
pthread_mutex_lock(&shm_ptr->order_mutex);	Lock shared memory buffer	
Order order = shm_ptr->iceCreamOrders[shm_ptr->order_out];	Take out order from shared buffer	order
shm_ptr->order_out = (shm_ptr->order_out + 1) % BUFFER_SIZE;	Increment order_out index	order_out
pthread_mutex_unlock(&shm_ptr->order_mutex);	Unlock shared memory buffer	
sem_post(sem_empty);	Signal that one empty slot is now available	
sem_wait(&machine1.empty);	Wait for empty slot in machine1 buffer	
pthread_mutex_lock(&machine1.mutex);	Lock machine1 buffer	
machine1.buffer[machine1.in] = order;	Insert order into machine1	machine1.buffer
machine1.in = (machine1.in + 1) % MACHINE_BUFFER_SIZE;	Increment machine1 in index	machine1.in
pthread_mutex_unlock(&machine1.mutex);	Unlock machine1 buffer	
sem_post(&machine1.full);	Signal that machine1 has a full slot	
sleep(1);	Pause briefly to simulate delay	
m->in = m->out = 0;	Initialize in and out to 0	m->in, m->out
sem_init(&m->full, 0, 0);	Initialize full semaphore to 0	m->full
sem_init(&m->empty, 0, MACHINE_BUFFER_SIZE);	Initialize empty semaphore	m->empty
pthread_mutex_init(&m->mutex, NULL);	Initialize mutex	m->mutex
sem_wait(&args->in_buf->full);	Wait for a full slot in input buffer	
pthread_mutex_lock(&args->in_buf->mutex);	Lock input buffer	
Order order = args->in_buf->buffer[args->in_buf->out];	Take order from input buffer	order
args->in_buf->out = (args->in_buf->out + 1) % MACHINE_BUFFER_SIZE;	Increment out index	args->in_buf->out
pthread_mutex_unlock(&args->in_buf->mutex);	Unlock input buffer	
sem_post(&args->in_buf->empty);	Signal an empty slot in input buffer	
if (strcmp(args->step, "Mixing") == 0) { ... }	Process Mixing step	order.isMixed
else if (strcmp(args->step, "Freezing") == 0) { ... }	Process Freezing step	order.isFrozen

else if (strcmp(args->step, "Packaging") == 0) { ... }	Process Packaging step	order.isPackaged
if (args->out_buf) { ... }	Send to next buffer if exists	args->out_buf
kill(order.customer_pid, SIGUSR1);	Signal customer that their order is ready	signal sent

Conclusion:

In conclusion, this project successfully demonstrates a robust implementation of inter-process and inter-thread synchronization using shared memory and POSIX semaphores. It highlights the importance of synchronization mechanisms such as **mutexes** (to protect critical sections) and **semaphores** (to manage producer-consumer relationships) in concurrent programming. The modular design allows for scalability, where multiple customer and machine processes can operate concurrently with minimal risk of data corruption or deadlocks. This system can serve as a foundation for more complex real-time production or simulation systems involving process coordination and resource sharing.