OPERATING SYSTEMS
WEEK 9 CODES

9.1 . The Tale of the Library Management System
PROGRAM :

```
import threading
import time
import random

class Book:
    def __init__(self, title):
        self.title = title
        self.is_available = True
        self.lock = threading.Lock()  # Each book gets its own lock

    def checkout(self):
        """Attempt to checkout the book, return False if already checked out."""
        with self.lock:
            if self.is_available:
                self.is_available = False
                return True
            else:
                return False

    def return_book(self):
        """Return the book, making it available."""
        with self.lock:
            if not self.is_available:
                self.is_available = True
                return True
            else:
                return False

class Library:
    def __init__(self):
        self.books = {}  # Store books by title
        self.book_lock = threading.Lock()  # Lock for managing overall inventory

    def add_book(self, book):
        """Add a book to the inventory."""
        with self.book_lock:
            self.books[book.title] = book
```

```
    def checkout_book(self, title):
        """Handle the checkout process."""
        with self.book_lock:
            if title in self.books:
                book = self.books[title]
                if book.checkout():
                    print(f"Book '{title}' checked out successfully.")
                    return True
                else:
                    print(f"Book '{title}' is already checked out.")
                    return False
            else:
                print(f"Book '{title}' not found in inventory.")
                return False

    def return_book(self, title):
        """Handle the return process."""
        with self.book_lock:
            if title in self.books:
                book = self.books[title]
                if book.return_book():
                    print(f"Book '{title}' returned successfully.")
                    return True
                else:
                    print(f"Book '{title}' was not checked out.")
                    return False
            else:
                print(f"Book '{title}' not found in inventory.")
                return False

    @staticmethod
    def simulate_user_action(library, action, book_title):
        """Simulate a user performing an action (checkout or return)"""
        time.sleep(random.uniform(0.5, 2))  # Simulate random delay in actions
        if action == "checkout":
            library.checkout_book(book_title)
        elif action == "return":
            library.return_book(book_title)

# Main program to test concurrency and locking
if __name__ == "__main__":
    # Create the library and some books
    library = Library()
    book1 = Book("Python Programming")
```

```
    book2 = Book("Data Structures and Algorithms")
    library.add_book(book1)
    library.add_book(book2)

    # Simulate concurrent user actions
    threads = []
    actions = ["checkout", "return"]

    for i in range(5):  # Simulating 5 users
        action = random.choice(actions)
        book_title = random.choice(["Python Programming", "Data Structures and Algorithms"])
        thread = threading.Thread(target=Library.simulate_user_action, args=(library, action,
book_title))
        threads.append(thread)
        thread.start()

    # Wait for all threads to complete
    for thread in threads:
        thread.join()

    print("Simulation finished.")
```

OUTPUT :

Book 'Data Structures and Algorithms' checked out successfully.
Book 'Data Structures and Algorithms' returned successfully.
Book 'Data Structures and Algorithms' was not checked out.
Book 'Data Structures and Algorithms' was not checked out.
Book 'Python Programming' was not checked out.
Simulation finished.

9.2 . The Tale of the Restaurant Reservation System

PROGRAM :

```
import threading
import time
import random

class Table:
    def __init__(self, table_id):
        self.table_id = table_id
        self.is_reserved = False
        self.version = 1  # Optimistic locking version
```

```
        self.lock = threading.Lock()  # Pessimistic locking for the table record

    def reserve_optimistic(self, customer_name):
        """Attempt to reserve the table using optimistic locking (check version)."""
        if self.is_reserved:
            print(f"Table {self.table_id} is already reserved. Reservation failed for
{customer_name}.")
            return False

        # Simulate a version check before reservation
        expected_version = self.version
        self.version += 1  # Increment version if reservation is successful
        self.is_reserved = True
        print(f"Table {self.table_id} successfully reserved for {customer_name} using optimistic
locking.")
        return True

    def reserve_pessimistic(self, customer_name):
        """Attempt to reserve the table using pessimistic locking (lock the table)."""
        with self.lock:  # Lock the table to prevent other threads from modifying it
            if self.is_reserved:
                print(f"Table {self.table_id} is already reserved. Reservation failed for
{customer_name}.")
                return False
            else:
                self.is_reserved = True
                print(f"Table {self.table_id} successfully reserved for {customer_name} using
pessimistic locking.")
                return True

    def release(self):
        """Release the table and mark it as available."""
        with self.lock:
            if self.is_reserved:
                self.is_reserved = False
                print(f"Table {self.table_id} is now available.")
            else:
                print(f"Table {self.table_id} is already available.")

class ReservationSystem:
    def __init__(self, num_tables):
        self.tables = [Table(i) for i in range(1, num_tables + 1)]  # Create tables 1 to num_tables
        self.system_lock = threading.Lock()  # Lock for the overall reservation system (to prevent
race conditions in system-wide actions)
```

```
    def book_table(self, customer_name, optimistic=False):
        """Attempt to book a table for a customer."""
        with self.system_lock:  # Lock the whole system for safety in managing multiple
reservations
            available_table = None
            for table in self.tables:
                if not table.is_reserved:
                    available_table = table
                    break

            if available_table:
                # Try to reserve the table using the chosen locking strategy
                if optimistic:
                    return available_table.reserve_optimistic(customer_name)
                else:
                    return available_table.reserve_pessimistic(customer_name)
            else:
                print(f"No available tables for {customer_name}. Please try again later.")
                return False

    def cancel_reservation(self, customer_name):
        """Cancel the reservation for the customer."""
        with self.system_lock:
            # Simulate finding the reservation (in a real system, we'd search a booking record)
            for table in self.tables:
                if table.is_reserved:
                    table.release()
                    print(f"Reservation canceled for {customer_name}.")
                    return True
            print(f"No reservation found for {customer_name}.")
            return False

    def handle_reservation(self, customer_name, optimistic=False):
        """Simulate handling the reservation transaction."""
        try:
            # Start the reservation process
            if not self.book_table(customer_name, optimistic):
                raise Exception("Reservation failed during the booking process.")

            # Simulate additional steps like notifying the customer, etc.
            time.sleep(random.uniform(0.5, 1.5))  # Simulate network delay or email sending

            # If no errors, finalize the reservation (commit the transaction)
```

```
            print(f"Reservation successfully processed for {customer_name}.")
            return True

        except Exception as e:
            print(f"Error during reservation for {customer_name}: {e}")
            # If any part of the reservation fails, roll back (release any reserved tables)
            self.cancel_reservation(customer_name)
            return False

# Simulate user interactions with the reservation system
def simulate_user_action(reservation_system, customer_name, optimistic=False):
    """Simulate a customer trying to reserve a table."""
    time.sleep(random.uniform(0.5, 2))  # Simulate random delay in user request
    reservation_system.handle_reservation(customer_name, optimistic)

# Main program to test concurrency and locking strategies
if __name__ == "__main__":
    # Create the reservation system with 10 tables
    reservation_system = ReservationSystem(num_tables=10)

    # Simulate concurrent reservation attempts
    customers = ["Alice", "Bob", "Charlie", "David", "Eve", "Frank", "Grace", "Hannah", "Isaac",
"Jack"]
    threads = []

    # Simulate customers using optimistic or pessimistic locking
    for customer in customers:
        action = random.choice(["optimistic", "pessimistic"])  # Randomly choose the locking
strategy
        optimistic = True if action == "optimistic" else False
        thread = threading.Thread(target=simulate_user_action, args=(reservation_system,
customer, optimistic))
        threads.append(thread)
        thread.start()

    # Wait for all threads to complete
    for thread in threads:
        thread.join()

    print("Reservation simulation completed.")
```

OUTPUT :

Table 1 successfully reserved for Eve using optimistic locking.
Table 2 successfully reserved for Isaac using pessimistic locking.
Table 3 successfully reserved for Alice using optimistic locking.
Table 4 successfully reserved for David using pessimistic locking.
Table 5 successfully reserved for Frank using optimistic locking.
Table 6 successfully reserved for Charlie using pessimistic locking.
Table 7 successfully reserved for Hannah using pessimistic locking.
Reservation successfully processed for Alice.
Table 8 successfully reserved for Jack using optimistic locking.
Table 9 successfully reserved for Bob using pessimistic locking.
Reservation successfully processed for Isaac.
Reservation successfully processed for Hannah.
Reservation successfully processed for Frank.
Reservation successfully processed for Jack.
Table 10 successfully reserved for Grace using optimistic locking.
Reservation successfully processed for David.
Reservation successfully processed for Charlie.
Reservation successfully processed for Eve.
Reservation successfully processed for Bob.
Reservation successfully processed for Grace.
Reservation simulation completed.

9.3 . The Tale of the Online Shopping Cart System

PROGRAM :

```
import threading
import time
import random

class InventoryItem:
    def __init__(self, item_id, name, quantity):
        self.item_id = item_id
        self.name = name
        self.quantity = quantity
        self.lock = threading.Lock()  # Lock for inventory updates

    def update_quantity(self, quantity_change):
        """Update inventory quantity and ensure consistency using pessimistic locking."""
        with self.lock:
            if self.quantity + quantity_change < 0:
                print(f"Not enough stock for {self.name}. Operation failed.")
                return False
            self.quantity += quantity_change
```

```
        print(f"Updated {self.name}: New quantity is {self.quantity}.")
        return True

class ShoppingCart:
    def __init__(self, cart_id):
        self.cart_id = cart_id
        self.items = {}  # Dictionary to hold items and their quantities
        self.lock = threading.Lock()  # Lock for shopping cart updates

    def add_item(self, item, quantity):
        """Add an item to the shopping cart, ensuring synchronization."""
        with self.lock:
            if item.item_id in self.items:
                self.items[item.item_id] += quantity
            else:
                self.items[item.item_id] = quantity
            print(f"Added {quantity} of {item.name} to cart {self.cart_id}. Current quantity:
{self.items[item.item_id]}")

    def remove_item(self, item, quantity):
        """Remove an item from the shopping cart, ensuring synchronization."""
        with self.lock:
            if item.item_id in self.items and self.items[item.item_id] >= quantity:
                self.items[item.item_id] -= quantity
                if self.items[item.item_id] == 0:
                    del self.items[item.item_id]
                print(f"Removed {quantity} of {item.name} from cart {self.cart_id}.")
            else:
                print(f"Not enough {item.name} in cart {self.cart_id} to remove.")

class ShopEase:
    def __init__(self, inventory):
        self.inventory = {item.item_id: item for item in inventory}
        self.cart_locks = {}  # Lock for each shopping cart
        self.transaction_lock = threading.Lock()  # Lock for managing the transaction process

    def process_transaction(self, cart, add_items, remove_items):
        """Process the entire transaction atomically."""
        with self.transaction_lock:
            # Step 1: Check if the cart is locked for another transaction
            if cart.cart_id in self.cart_locks:
                print(f"Cart {cart.cart_id} is currently being modified. Transaction aborted.")
                return False
```

```python
        # Lock the cart for this transaction
        self.cart_locks[cart.cart_id] = cart.lock

        try:
            # Step 2: Handle adding items to the cart
            for item, quantity in add_items.items():
                if not self.inventory[item.item_id].update_quantity(-quantity):
                    print("Transaction failed: Not enough inventory.")
                    return False
                cart.add_item(item, quantity)

            # Step 3: Handle removing items from the cart
            for item, quantity in remove_items.items():
                cart.remove_item(item, quantity)
                self.inventory[item.item_id].update_quantity(quantity)

            # Commit transaction (all steps succeeded)
            print(f"Transaction for cart {cart.cart_id} completed successfully.")
            return True

        except Exception as e:
            print(f"Error during transaction: {e}")
            return False

        finally:
            # Release the cart lock after transaction is complete
            del self.cart_locks[cart.cart_id]

# Example usage
def simulate_customer_action(shop, cart, add_items, remove_items):
    """Simulate a customer interacting with the shopping cart."""
    time.sleep(random.uniform(0.5, 2))  # Random delay to simulate user action
    success = shop.process_transaction(cart, add_items, remove_items)
    if success:
        print(f"Transaction for cart {cart.cart_id} was successful.")
    else:
        print(f"Transaction for cart {cart.cart_id} failed.")

if __name__ == "__main__":
    # Create inventory with 5 items
    inventory = [
        InventoryItem(1, "Laptop", 10),
        InventoryItem(2, "Headphones", 20),
        InventoryItem(3, "Keyboard", 30),
```

```
        InventoryItem(4, "Mouse", 15),
        InventoryItem(5, "Monitor", 5),
    ]

    # Create the ShopEase system with the inventory
    shop = ShopEase(inventory)

    # Create shopping carts for 3 customers
    cart1 = ShoppingCart(1)
    cart2 = ShoppingCart(2)
    cart3 = ShoppingCart(3)

    # Simulate customer actions concurrently
    threads = []
    threads.append(threading.Thread(target=simulate_customer_action, args=(shop, cart1,
{inventory[0]: 2}, {inventory[1]: 1})))
    threads.append(threading.Thread(target=simulate_customer_action, args=(shop, cart2,
{inventory[1]: 3}, {inventory[3]: 1})))
    threads.append(threading.Thread(target=simulate_customer_action, args=(shop, cart3,
{inventory[2]: 5}, {inventory[0]: 1})))

    # Start all threads
    for thread in threads:
        thread.start()

    # Wait for all threads to complete
    for thread in threads:
        thread.join()

    print("Shopping simulation completed.")
```

OUTPUT :

```
Updated Laptop: New quantity is 8.
Added 2 of Laptop to cart 1. Current quantity: 2
Not enough Headphones in cart 1 to remove.
Updated Headphones: New quantity is 21.
Transaction for cart 1 completed successfully.
Transaction for cart 1 was successful.
Updated Keyboard: New quantity is 25.
Added 5 of Keyboard to cart 3. Current quantity: 5
Not enough Laptop in cart 3 to remove.
Updated Laptop: New quantity is 9.
Transaction for cart 3 completed successfully.
Transaction for cart 3 was successful.
```

Updated Headphones: New quantity is 18.
Added 3 of Headphones to cart 2. Current quantity: 3
Not enough Mouse in cart 2 to remove.
Updated Mouse: New quantity is 16.
Transaction for cart 2 completed successfully.
Transaction for cart 2 was successful.
Shopping simulation completed.

9.4 . The Tale of the Collaborative Document Editing System
PROGRAM :

```
import threading
import time
import random

class Document:
    def __init__(self, content):
        """Initialize the document with sections and locks for each section."""
        self.sections = content.split("\n")
        self.locks = [threading.Lock() for _ in self.sections]  # One lock per section
        self.version = [0] * len(self.sections)  # Versioning to track changes for each section

    def get_section(self, section_id):
        """Get the content of a specific section."""
        return self.sections[section_id]

    def update_section(self, section_id, new_content, version):
        """Attempt to update a section if the version matches (optimistic locking)."""
        with self.locks[section_id]:
            if self.version[section_id] != version:
                print(f"Conflict detected in section {section_id}: Version mismatch (expected {version},
found {self.version[section_id]}).")
                return False
            # If no conflict, update the section and increment the version
            self.sections[section_id] = new_content
            self.version[section_id] += 1
            print(f"Section {section_id} updated successfully by User {self.user_id}. New version:
{self.version[section_id]}")
            return True

    def lock_section(self, section_id):
        """Lock a section to prevent other users from editing it."""
        self.locks[section_id].acquire()

    def unlock_section(self, section_id):
```

```
        """Unlock a section after editing."""
        self.locks[section_id].release()

class User:
    def __init__(self, user_id, document):
        self.user_id = user_id
        self.document = document
        self.edits = {}  # Track edits for each section

    def edit_section(self, section_id, new_content):
        """Simulate editing a document section."""
        # Get the current version of the section
        current_version = self.document.version[section_id]

        print(f"User {self.user_id} is editing section {section_id}, current version: {current_version}")

        # Lock the section (pessimistic locking)
        self.document.lock_section(section_id)
        try:
            # Simulate some editing delay
            time.sleep(random.uniform(0.5, 1.5))
            # Update the section with the new content
            if not self.document.update_section(section_id, new_content, current_version):
                print(f"User {self.user_id} failed to update section {section_id} due to a conflict.")
                return False
            self.edits[section_id] = (new_content, current_version + 1)
            print(f"User {self.user_id} successfully edited section {section_id}.")
            return True
        finally:
            # Unlock the section after editing
            self.document.unlock_section(section_id)

    def attempt_conflict_resolution(self, section_id):
        """Attempt to resolve a conflict by asking the user to merge changes."""
        print(f"User {self.user_id} is resolving a conflict in section {section_id}.")
        # Simulate the user resolving the conflict manually.
        time.sleep(1)
        print(f"User {self.user_id} resolved conflict in section {section_id}.")
        return True

class EditTogether:
    def __init__(self, initial_content):
        self.document = Document(initial_content)
        self.print_lock = threading.Lock()  # Lock for thread-safe printing
```

```
    def start_editing_session(self, user, section_id, new_content):
        """Start a new editing session for a user."""
        success = user.edit_section(section_id, new_content)
        if not success:
            # If the update failed, resolve conflicts if necessary
            user.attempt_conflict_resolution(section_id)

    def thread_safe_print(self, msg):
        """Print with thread safety."""
        with self.print_lock:
            print(msg)

# Example usage
def simulate_user_editing(edit_together, user, section_id, new_content):
    """Simulate a user trying to edit a document section."""
    edit_together.start_editing_session(user, section_id, new_content)

if __name__ == "__main__":
    # Initial document content with 5 sections
    document_content = """This is the first section of the document.
This is the second section of the document.
This is the third section of the document.
This is the fourth section of the document.
This is the fifth section of the document."""

    # Initialize the collaborative document editing system
    edit_together = EditTogether(document_content)

    # Create users
    user1 = User(1, edit_together.document)
    user2 = User(2, edit_together.document)

    # Simulate concurrent edits to the same section
    threads = []
    threads.append(threading.Thread(target=simulate_user_editing, args=(edit_together, user1,
1, "User 1 edited the second section.")))
    threads.append(threading.Thread(target=simulate_user_editing, args=(edit_together, user2,
1, "User 2 edited the second section.")))

    # Start all threads (simulating concurrent editing)
    for thread in threads:
        thread.start()
```

```
    # Wait for all threads to complete
    for thread in threads:
        thread.join()

    edit_together.thread_safe_print("Document editing simulation completed.")
```

OUTPUT :

```
User 1 is editing section 1, current version: 0
User 2 is editing section 1, current version: 0
User 1 failed to update section 1 due to a conflict.
User 2 successfully edited section 1.
User 2 resolved conflict in section 1.
User 1 failed to update section 1 due to a conflict.
User 2 successfully edited section 1.
Document editing simulation completed.
```

9.5 . The Tale of the Bank Account Management System

PROGRAM :

```
import threading
import time
import random

# Account class to represent each customer's account
class Account:
    def __init__(self, account_id, initial_balance=0):
        self.account_id = account_id
        self.balance = initial_balance
        self.lock = threading.Lock()  # Lock for pessimistic locking
        self.version = 0  # Version number for optimistic locking

    def deposit(self, amount, version):
        """Deposit money into the account with optimistic locking."""
        with self.lock:
            if self.version != version:
                print(f"Conflict detected: Account {self.account_id} balance changed during
transaction.")
                return False
            self.balance += amount
            self.version += 1
            print(f"Deposited {amount} into Account {self.account_id}. New balance: {self.balance}")
            return True

    def withdraw(self, amount, version):
        """Withdraw money from the account with optimistic locking."""
```

```
        with self.lock:
            if self.version != version:
                print(f"Conflict detected: Account {self.account_id} balance changed during
transaction.")
                return False
            if self.balance < amount:
                print(f"Insufficient funds in Account {self.account_id}. Withdrawal failed.")
                return False
            self.balance -= amount
            self.version += 1
            print(f"Withdrew {amount} from Account {self.account_id}. New balance: {self.balance}")
            return True

    def get_balance(self):
        """Get the current balance of the account."""
        return self.balance

# Bank class to manage multiple accounts and transactions
class Bank:
    def __init__(self):
        self.accounts = {}  # A dictionary to store accounts by their account_id
        self.lock = threading.Lock()  # Lock to manage access to the accounts dictionary

    def add_account(self, account_id, initial_balance=0):
        """Add a new account to the bank."""
        with self.lock:
            self.accounts[account_id] = Account(account_id, initial_balance)

    def get_account(self, account_id):
        """Get an account by its ID."""
        with self.lock:
            return self.accounts.get(account_id)

    def transfer(self, from_account_id, to_account_id, amount):
        """Transfer money from one account to another."""
        from_account = self.get_account(from_account_id)
        to_account = self.get_account(to_account_id)

        if from_account and to_account:
            # We use the version of the source account for optimistic locking
            version = from_account.version

            # Try to withdraw from the source account
            if not from_account.withdraw(amount, version):
```

```
                print(f"Transaction failed: Unable to withdraw {amount} from Account
{from_account_id}.")
                return False

            # Try to deposit into the destination account
            if not to_account.deposit(amount, version):
                # If deposit fails, rollback the withdrawal (atomic transaction)
                print(f"Transaction failed: Unable to deposit {amount} into Account {to_account_id}.
Rolling back.")
                from_account.deposit(amount, version)  # Rollback the withdrawal
                return False

            print(f"Transaction successful: {amount} transferred from Account {from_account_id} to
Account {to_account_id}.")
            return True
        return False

# Simulate a bank system with multiple users
def simulate_transactions(bank, from_account_id, to_account_id, amount):
    """Simulate multiple transactions happening concurrently."""
    success = bank.transfer(from_account_id, to_account_id, amount)
    if not success:
        print(f"Transaction failed for {from_account_id} to {to_account_id} with amount {amount}.")

if __name__ == "__main__":
    # Initialize the bank system
    bank = Bank()

    # Create some accounts with initial balances
    bank.add_account("A001", 1000)
    bank.add_account("A002", 1500)

    # Start several threads to simulate concurrent transactions
    threads = []
    threads.append(threading.Thread(target=simulate_transactions, args=(bank, "A001", "A002",
200)))
    threads.append(threading.Thread(target=simulate_transactions, args=(bank, "A002", "A001",
300)))
    threads.append(threading.Thread(target=simulate_transactions, args=(bank, "A001", "A002",
100)))
    threads.append(threading.Thread(target=simulate_transactions, args=(bank, "A002", "A001",
50)))

    # Start all threads (simulating concurrent transactions)
```

```
    for thread in threads:
        thread.start()

    # Wait for all threads to complete
    for thread in threads:
        thread.join()

    # Final balances after all transactions
    print(f"Final balance of Account A001: {bank.get_account('A001').get_balance()}")
    print(f"Final balance of Account A002: {bank.get_account('A002').get_balance()}")
```

OUTPUT :

Withdrew 200 from Account A001. New balance: 800
Deposited 200 into Account A002. New balance: 1700
Transaction successful: 200 transferred from Account A001 to Account A002.
Withdrew 300 from Account A002. New balance: 1400
Deposited 300 into Account A001. New balance: 1100
Transaction successful: 300 transferred from Account A002 to Account A001.
Withdrew 100 from Account A001. New balance: 1000
Deposited 100 into Account A002. New balance: 1500
Transaction successful: 100 transferred from Account A001 to Account A002.
Withdrew 50 from Account A002. New balance: 1450
Deposited 50 into Account A001. New balance: 1050
Transaction successful: 50 transferred from Account A002 to Account A001.
Final balance of Account A001: 1050
Final balance of Account A002: 1450