Assignment\_26-08-2024

Author: ThanhTH10

Date: 27/08/2024

1. Implement move operation as data member int \*p;

#include <iostream>

#include <utility>  // For std::move

class MyClass {

private:

    int \*p;  // Pointer to an integer

public:

    // Default constructor

    MyClass() : p(new int(0)) {

        std::cout << "Default constructor called\n";

    }

    // Parameterized constructor

    MyClass(int value) : p(new int(value)) {

        std::cout << "Parameterized constructor called\n";

    }

    // Copy constructor

    MyClass(const MyClass& other) : p(new int(\*other.p)) {

        std::cout << "Copy constructor called\n";

    }

    // Move constructor

    MyClass(MyClass&& other) noexcept : p(other.p) {

        other.p = nullptr;  // Transfer ownership, leaving other with a nullptr

        std::cout << "Move constructor called\n";

    }

    // Copy assignment operator

    MyClass& operator=(const MyClass& other) {

        if (this == &other) return \*this;  // Self-assignment check

        delete p;  // Clean up existing resource

        p = new int(\*other.p);  // Allocate new memory and copy the value

        std::cout << "Copy assignment operator called\n";

        return \*this;

    }

    // Move assignment operator

    MyClass& operator=(MyClass&& other) noexcept {

        if (this == &other) return \*this;  // Self-assignment check

        delete p;  // Clean up existing resource

        p = other.p;  // Transfer ownership

        other.p = nullptr;  // Leave other with a nullptr

        std::cout << "Move assignment operator called\n";

        return \*this;

    }

    // Destructor

    ~MyClass() {

        delete p;  // Clean up memory

        std::cout << "Destructor called\n";

    }

    // Function to set value

    void setValue(int value) {

        \*p = value;

    }

    // Function to get value

    int getValue() const {

        return \*p;

    }

};

int main() {

    MyClass a(10);  // Parameterized constructor

    MyClass b = std::move(a);  // Move constructor

    MyClass c;

    c = std::move(b);  // Move assignment operator

    std::cout << "Value of c: " << c.getValue() << '\n';

    std::cout << "Value of a: " << (a.getValue()) << '\n';  // This will be undefined behavior

    return 0;

}

2. what is process and state diagram

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Process Diagram** | **State Diagram** |
| **Purpose** | Represents the sequence of steps or activities in a process. | Represents the different states of an object or system and transitions between them. |
| **Focus** | Workflow and steps in a process. | States and transitions of an object or system. |
| **Elements** | - Start/End points | - States |
| - Process steps | - Transitions |
| - Decisions | - Events |
| - Inputs/Outputs | - Actions |
| - Arrows |  |
| **Usage** | Business process modeling, workflow management, software development. | Modeling dynamic behavior of objects, especially in software engineering. |
| **Example** | A flowchart of order processing: | A state diagram for a traffic light: |
| - Start -> Receive Order -> Check Inventory -> Process Payment -> Ship Product -> End | - Red -> Green -> Yellow -> Red |
| **Diagram Type** | Flowchart, activity diagram | State machine diagram, state transition diagram |

3. Round robin scheduling – Explore

Round Robin Scheduling is a CPU scheduling algorithm designed to fairly allocate CPU time among all processes. In this approach, each process is assigned a fixed time slice or quantum. When a process’s quantum expires, it is moved to the end of the queue, and the CPU scheduler picks the next process in line. This method ensures that all processes get a chance to execute and prevents any single process from monopolizing the CPU. However, while Round Robin is fair and simple to implement, it can lead to significant context switching overhead if the time quantum is too short, and it may not efficiently handle processes with widely varying burst times or CPU-bound processes. The effectiveness of Round Robin depends on selecting an appropriate time quantum and may require adjustments to optimize performance for specific workloads.

4. What is producer and consumer problem.

The **Producer-Consumer Problem** is a classic example of a synchronization issue in concurrent programming, involving two types of processes:

1. **Producer**: Creates and places items into a shared buffer.
2. **Consumer**: Retrieves and processes items from the shared buffer.

**Core Challenges**

1. **Buffer Management**:
   * **Overflow**: The producer must wait if the buffer is full.
   * **Underflow**: The consumer must wait if the buffer is empty.
2. **Synchronization**:
   * Ensuring that the producer and consumer do not access the buffer simultaneously in a way that causes conflicts or data corruption.

**Objective**

To ensure smooth coordination between the producer and consumer so that:

* The producer does not add items when the buffer is full.
* The consumer does not attempt to remove items when the buffer is empty.

**Typical Solutions**

1. **Semaphores**:
   * **Empty Semaphore**: Counts available slots in the buffer.
   * **Full Semaphore**: Counts filled slots in the buffer.
   * **Mutex**: Ensures mutual exclusion when accessing the buffer.
2. **Condition Variables**: Used for signaling between producer and consumer when the buffer is full or empty.

5. What is dead lock with mutex

**Deadlock** with mutexes occurs when processes are stuck waiting for each other to release resources, leading to a standstill. For a deadlock to happen, the following conditions must be met:

1. **Mutual Exclusion**: At least one resource is held exclusively by one process.
2. **Hold and Wait**: A process holding a resource is waiting for additional resources held by other processes.
3. **No Preemption**: Resources cannot be forcibly taken from processes.
4. **Circular Wait**: A circular chain of processes exists, each waiting for a resource held by the next process in the chain.

**Example**:

* **Process A** locks **Mutex1** and waits for **Mutex2**.
* **Process B** locks **Mutex2** and waits for **Mutex1**.

Both processes are in deadlock because each is holding a mutex the other needs.