Semaphore Exercise: Resource Pool

Overview

This exercise demonstrates C++20 `std::counting_semaphore` through a practical **Resource Pool** implementation - a common pattern in systems programming.

Key Concepts Covered

1. Counting Semaphore Basics

```
std::counting_semaphore<> available_resources_{pool_size};

// Acquire (decrement) - blocks if count is 0
available_resources_.acquire();

// Release (increment) - wakes up waiting threads
available_resources_.release();

// Try with timeout
bool success = available_resources_.try_acquire_for(timeout);
```

2. Real-World Application

- **Database Connection Pool**: Limit concurrent database connections
- **Thread Pool**: Limit number of active worker threads

- **File Handle Pool**: Prevent file descriptor exhaustion
- **Network Connection Pool**: Manage HTTP/TCP connections

3. Semaphore vs Mutex

Aspect	Mutex	Semaphore
Purpose	Mutual exclusion (binary)	Resource counting
Max holders	1	N (configurable)
Use case	Protect critical sections	Limit resource access
Blocking	Only if locked	Only if count = 0

4. Advanced Features Demonstrated

Timeout Operations

```
// Non-blocking attempt with timeout
auto resource = pool.try_acquire(std::chrono::milliseconds(100));
if (!resource) {
    // Handle timeout - resource not available
}
```

RAII Resource Management

```
{
    ResourceGuard<Connection> guard(pool);  // Auto-acquire
    guard->execute_query("SELECT * FROM users");
    // Auto-release when guard goes out of scope
}
```

Statistics Tracking

- **Peak usage**: Maximum concurrent resources in use
- **Total acquisitions/releases**: Lifetime counters
- **Current usage**: Active resource count

Interview Relevance

Common Questions

1. **"How would you implement a connection pool?"**
   - Show semaphore-based limiting
   - Demonstrate timeout handling
   - Explain RAII for safety
2. **"What's the difference between semaphore and condition variable?"**
   - Semaphore: Built-in counter, simpler for resource limiting
   - Condition variable: More flexible, requires manual state management
3. **"How do you prevent resource leaks?"**
   - RAII wrappers (ResourceGuard)
   - Exception safety
   - Automatic cleanup

Production Considerations

- **Thread safety**: All operations are thread-safe
- **Exception safety**: RAII ensures cleanup
- **Performance**: Atomic operations for statistics
- **Scalability**: Lock-free semaphore operations

Test Scenarios Covered

1. **BasicAcquireRelease**: Simple acquire/release cycle
2. **RAIIWrapper**: Automatic resource management
3. **PoolExhaustion**: Behavior when resources depleted
4. **ConcurrentAccess**: Multiple threads competing for resources

5. **SemaphoreBlocking**: Proper blocking/unblocking behavior

Key Takeaways

Semaphores Excel At:

- **Resource limiting**: Natural counting mechanism
- **Producer-consumer**: Alternative to condition variables
- **Rate limiting**: Control access frequency
- **Load balancing**: Distribute work across resources

Design Patterns

- **Resource pooling**: Pre-allocate expensive resources
- **Throttling**: Limit concurrent operations
- **Backpressure**: Prevent system overload

This exercise shows how semaphores provide an elegant solution for resource management - a critical skill for backend systems and high-performance applications!