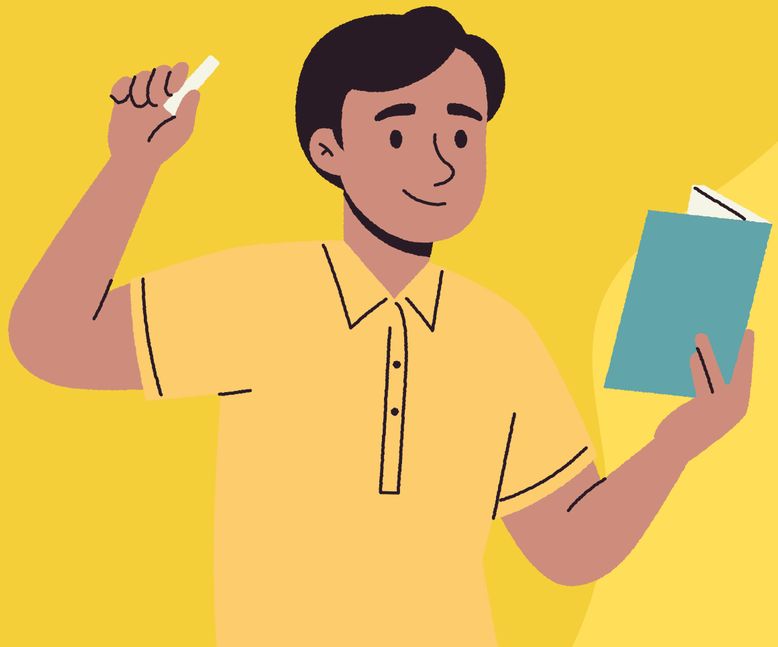
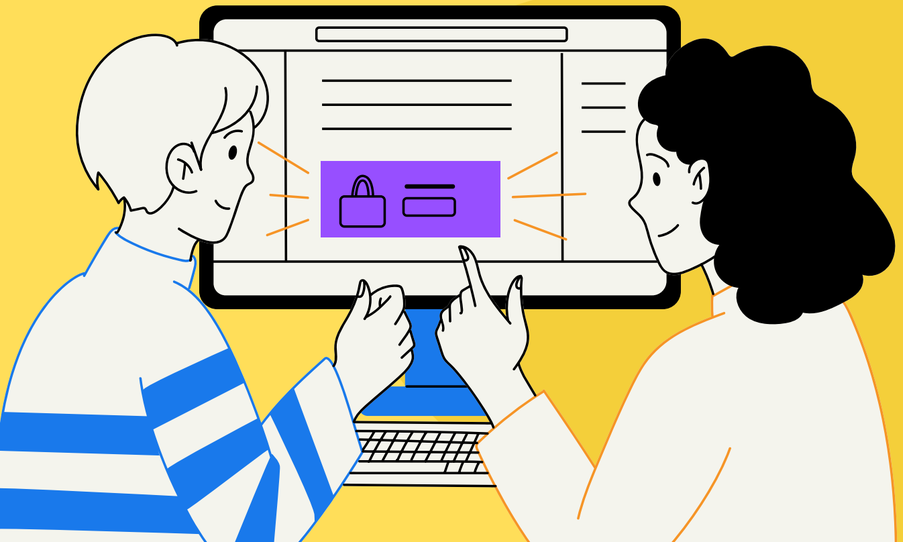
Module 6

ADVANCED CONCURRENCY AND SYNCHRONIZATION

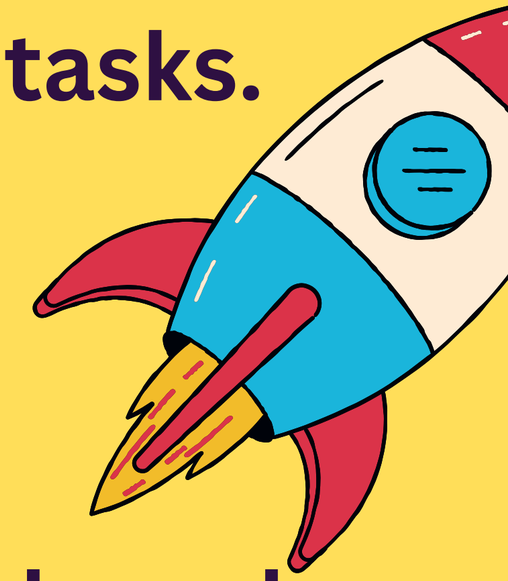


SYNCHRONIZATION TECHNIQUES



sync.WaitGroup

Used to wait for a group of goroutines to finish executing. Helps coordinate concurrent tasks.



sync.Mutex

A mutual exclusion lock used to protect shared resources from concurrent access.

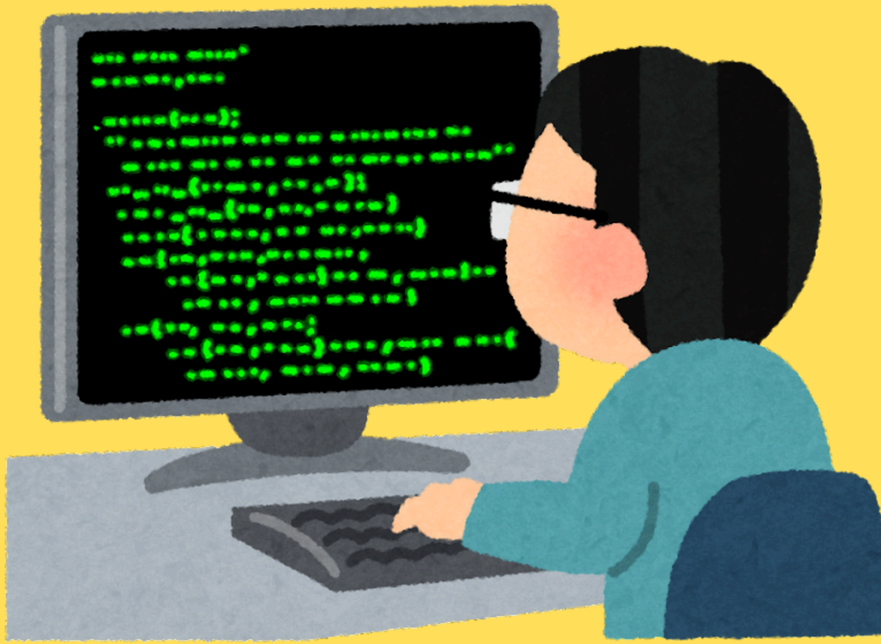
sync.RWMutex

A read/write mutex that allows multiple readers but only one writer at a time.

Key Points to Remember

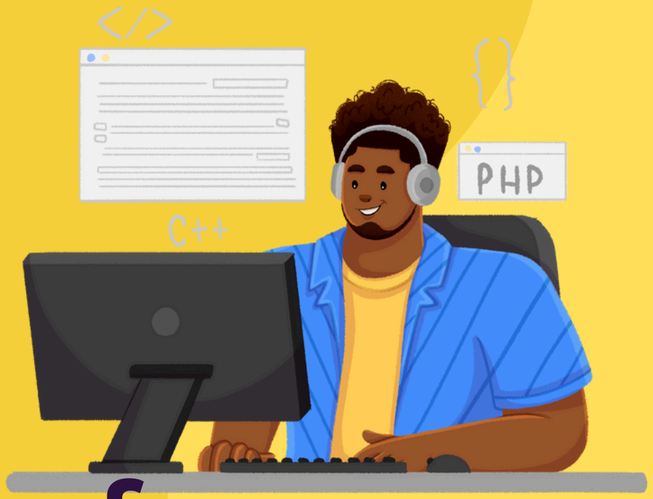
- Deadlocks occur when goroutines wait indefinitely due to cyclic dependencies on locks.
- Race Conditions happen when multiple goroutines access and modify shared resources without synchronization.
- Always unlock a sync.Mutex or sync.RWMutex after locking to avoid deadlocks (defer m.Unlock() is a best practice).

WORKER POOLS AND PIPELINES



Worker Pool

A pattern where a fixed number of goroutines process a large number of tasks from a queue (channel).



Pipeline

A chain of stages where data flows from one stage to another via channels.

Key Points to Remember

- Avoid excessive goroutines: Spawning too many goroutines without management can lead to high memory usage and scheduling overhead.
- Use buffered channels: Helps manage task queue size and prevents blocking.
- Graceful shutdown: Always close channels properly to prevent goroutines from getting stuck.

CONTEXT PACKAGE

context.Background()

The root context, used as a base for other contexts.

context.WithCancel()

Creates a child context that can be canceled explicitly.

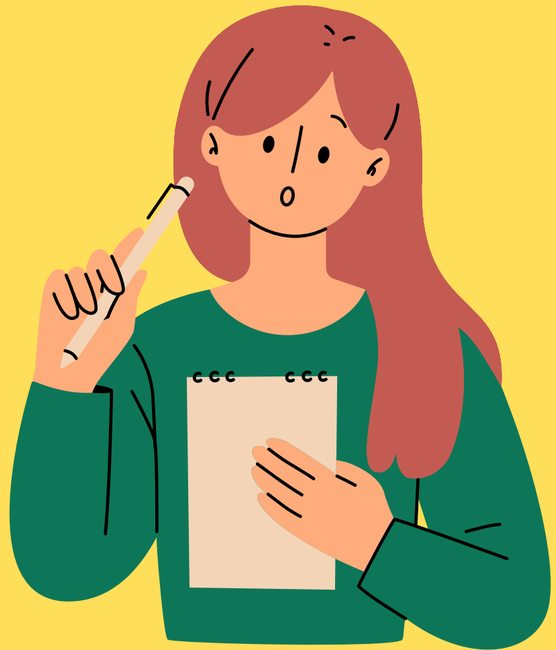
context.WithDeadline()

Sets an absolute time limit after which the context expires.

Key Points to Remember

- Use context.WithCancel() to stop goroutines gracefully: Always pass ctx down to goroutines and check ctx.Done().
- Avoid goroutine leaks: Ensure that goroutines properly exit when the context is canceled.
- Set timeouts for network calls to prevent hanging operations (e.g., context.WithTimeout() for HTTP requests).

REAL-WORLD CONCURRENCY PATTERNS



Producer-Consumer

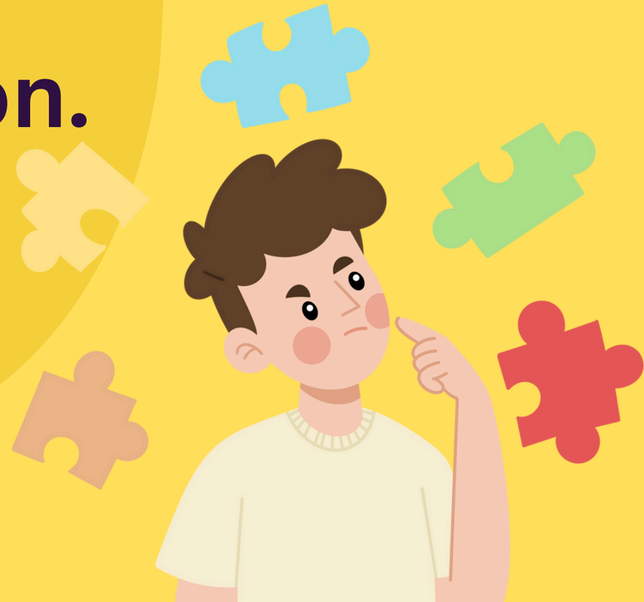
- Producers generate data; consumers process it.
- Uses a queue (buffer) for decoupling.
- Requires synchronization (locks, semaphores).
- Used in job scheduling, logging, and message queues.

Fan-In

- Multiple producers → One consumer (merging data streams).

- Reduces contention, aggregates results efficiently.

Used in data processing, event aggregation.



Fan-Out

- One producer → Multiple consumers (distributing tasks).
- Improves parallel processing and system scalability.
- Used in load balancing, task distribution (e.g., web servers).

THANK YOU