COSC 3750 — EXAM 1 STUDY GUIDE

What "System Programming" Mean (Big Picture)

- System programming is writing programs that talk directly to the OS:
  - Manage
    - Files
    - Processes
    - Permissions
    - Signals
    - Terminals
    - Network Connections
  - Use system calls (kernel interfaces) rather than just high-level libraries
  - Care about resources, errors, and edge cases
- Why Go Fits Well
  - Strong standard library
    - Files, Net, OS, IO
  - Explicit error handling → fewer failures
  - Goroutines & channels make concurrency safer than raw threads
- System Calls Vs. Library Calls
  - System Calls
    - Transition from user space → kernel
  - libc / Go Wrappers Exist For:
    - Portability, Convenience, Error Translation
  - Not every function you call is a system call

Go Basics for System Code (CH. 1-2)

- Toolchain
  - go build, go run, go test
- *Modules* manage dependencies
  - go.mod
- Code is organized in *packages*
- Idioms you MUST expect:
  - Always checks errors
  - Small, focused functions
  - Interfaces define behavior, not data

Files and File I/O (GO Ch. 3 + TLPI Ch. 4-5)

- Core Model
  - Files are accessed through file descriptors
  - Almost everything is treated as a file
    - Files, Pipes, Sockets, Terminals → treated like files
- Key Rules
  - open → read/write → close
  - reads and writes may be partial → loop until done
  - Always close resources
    - fd leaks = real bugs
- WHY Reads/Writes Happen
  - Pipes & Sockets
  - Nonblocking I/O
  - Kernel Buffering Limits
  - Signals Interrupting Syscalls
- Metadata
  - Files have:
    - size, permissions, owner, timestamps
  - A *filename* is just a directory entry, NOT the file itself

Permissions & Filesystems (Go Ch. 4 + TLPI Ch. 14-15)

- Permissions
  - Format
    - rwxrwxrwx (user / group / other)
  - umask
    - removes permissions from newly created files
  - *chmod* changes permissions | *chown* changes ownership
- Filesystems
  - A filesystem = data + metadata
  - Inode
    - Holds metadata
      - Permissions, owner, size
  - Filenames
    - Point to inodes → name ≠ file
  - Mounting
    - Merges filesystems into one directory tree

Directories & Links (TLPI Ch. 18)

- Directories map names → inode numbers
- Hard Link
  - Another name for the same inode
- Symbolic Link
  - A file that points to a pathname
- Deleting a file
  - unlink
    - Removes a name
  - Data is freed only when link count = 0 AND file not open

Go I/O Interfaces (Go Ch. 5)

- Stream Mental Model
  - Treat everything as a system of bytes
  - io.Reader / io.Writer
    - Unify files, sockets, buffers
- Composition
  - You can wrap streams
    - Buffering
    - Limiting
    - Tee (Copy to Multiple Outputs)
  - This enables powerful pipelines with minimal code

Concurrency (Go Ch. 6)

- Goroutines
  - Lightweight concurrent functions
- Channels
  - Communication & Synchronization
- Use concurrency when
  - You need parallel work
  - You need to wait on multiple events
- Still need coordination for
  - Ordering
  - Cancellation
  - Limited Resources
- Goroutines ≠ OS Threads
  - Go schedules goroutines onto threads
  - Much cheaper to create and manage

Signals (Go Ch. 7 + TLPI Ch. 20-22)

- What Signals Are
  - Asynchronous Notifications
    - Ctrl-C, Timers, Child Exit
  - Default Actions
    - Terminate, Stop, Ignore, Continue
- Signals Can Interrupt:
  - read, write, accept
- Programs MUST:
  - Retry OR handle EINTR
- Important Concepts
  - Generated Vs. Delivered
  - Pending Vs. Handling
  - Signal Masks Block Delivery
- Most functions are NOT async-signal-safe
- Handlers
  - Interrupt normal execution
  - MUST be minimal & async-signal-safe
  - In GO
    - Usually delivered through channels → clean shutdown paths

Processes & Execution (TLPI Ch. 24-35)

- fork / exec Model
  - fork(0
    - Child is a copy
    - Same memory, same fds
  - exec()
    - Replace program image, Same PID
- Termination
  - Processes exit with a status
  - Parent must wait() or waitpid() to avoid zombies
- Why wait() Matters
  - Without wait()
    - Zombie processes accumulate
  - With wait()
    - Reaps Child
    - Retrieves Exit Status
- Zombies vs. Orphans
  - Zombie: exited, not waited on
  - Orphan: parent exited; adopted by init/systemd

Process Groups, Job Control (TLPI Ch. 34)

- Processes belong to process groups
- Groups belong to sessions
- Terminal send signals (Ctrl-C) to the foreground process group
- Job control exists so shells can manage foreground/background jobs

Networking Basics (Go Ch. 8)

- Client/Server Model
- Common Pattern:
  - Listen → Accept → Read/Write → Close
- Go's net package abstracts sockets cleanly

Daemons & Logging (TPLI Ch. 37)

- Daemons:
  - Run in background
  - No controlling terminal
  - Clean environment
    - cwd, umask, file descriptors
- Logging goes to log files or syslog, NOT stdout

Terminals (TLPI Ch. 62)

- Terminal Driver Model
  - Input/Output Queues & Line Discipline
  - Terminal interprets special characters
- Canonical Vs. Noncanonical
  - Canonical:
    - line-at-a-time
    - Enter REQUIRED
  - Noncanonical:
    - character-at-a-time
    - Used by editors, games
- Special Characters
  - INTR → Ctrl-C → SIGINT
  - EOF → Ctrl-D
  - SUSP → Ctrl-Z
- Termios
  - tcgetattr / tcsetattr control terminal behavior
  - Tools like stty modify these flags

Pseudoterminals (TLPI Ch. 64)

- pty slave
  - Looks like a real terminal
- pty master
  - Controlled by another program
- Used by
  - Shells, ssh, script, terminal emulators
- WHY SSH, Shells, Scripts Need PTYs
  - Programs behave differently when stdin is a terminal
  - Line editing, echoing, signals require terminal semantics
  - ptys fake a real terminal
- Typical Pattern
  1. Open pty master
  2. fork
  3. Child: New session → open slave → dup to stdin/out/err → exec
  4. Parent: relay data between user and child

Pipes (TLPI Ch. 44)

- Unidirectional byte streams
- Used heavily by shells
- EOF happens when write end is closed
  - Occur only when ALL write ends are closed
- Deadlocks happen when both sides wait forever
- Deadlocks Happens When:
  - One process waits to read
  - Other waits to write
  - Neither closes unused pipe ends
- ALWAYS close unused ends after fork to avoid deadlocks
- Classic Pattern
  - pipe → fork → dup → exec

Exam Habits to Remember

- Assume partial reads/writes
- Always know what must be closed
- FIles, Pipes, Sockets, Terminals ALL share the same I/O model
- Errors are normal → handle them
- Think in terms of resources and lifetimes

File Descriptor Flags Vs File Status Flags

- File Descriptor Flags
  - Per descriptor
  - Ex: close-on-exec
- File Status Flags
  - Per open file
  - EX: Append, Nonblocking
- They affect behavior differently & live in different places

Mental Model That Solves Most Questions – Ask Yourself

- What resource is created?
- Who owns it?
- Who must close it?
- What happens if it isn't closed?