Course Overview and The Shell

CMSC398W: Practical Tools For Efficient Development

Mohammad Durrani

June 22, 2026

The Core Problem

Software does exactly what you tell it, not what you intend.

- Debugging = bridging the gap between intended vs. actual behavior
- Though time-intensive, systematic debugging saves time in the long run
- Instead of a reactive slowdown, structured debugging accelerates development

A Systematic Approach: Observe, Hypothesize, Test

Debugging is methodical, not trial-and-error.
Five steps:

1. Reproduce consistently → If you can't reproduce, you can't fix
2. Isolate the problem → Use binary search strategy in code
3. Form a hypothesis → Develop a theory based on symptoms
4. Test the hypothesis → Only change one thing at a time
5. Fix and verify → Confirm it solves the issue

Example Walkthrough

Web app crashes on file upload.

- Step 1: Reproduce with user uploads
- Step 2: Isolate $\rightarrow$ file parsing function
- Step 3: Hypothesize $\rightarrow$ large files cause memory issue
- Step 4: Test $\rightarrow$ try different file sizes
- Step 5: Fix $\rightarrow$ switch to streaming instead of full load

Print Debugging: The Foundation

Brian Kernighan (1979): *"The most effective debugging tool is still careful thought, coupled with judiciously placed print statements."*

- Print statements = simplest debugging method
- Iterate by adding prints near problem areas until root cause is clear
- Preferred because it's quick, easy, and effective

Logging: Enhanced Print Debugging

Logging advantages:

- Output to files, sockets, remote servers (easier to review than terminal)
- Severity levels: `INFO`, `DEBUG`, `WARN`, `ERROR`
- Long-term monitoring + immediate debugging
- Structured logging (keyvalue pairs) $\rightarrow$ searchable, filterable, analyzable

Logging Best Practices

Practice	Purpose
Set log levels properly	Filter noise
Use structured logging	Keyvalue pairs for analysis
Ensure traceability	Unique messages/prefixes to source
Use correlation IDs	Track requests end-to-end
Be strategic	Don't let logging consume resources
Never log sensitive info	Security first

External Logging

When working with external dependencies (web servers, databases, containers), check their logs:

- Most programs write to `/var/log` on UNIX systems
- Containerized apps: `docker logs <container-name>`
- Client-side error messages often lack detail

When Print Debugging Is Not Enough

Debuggers are programs that let you interact with execution, allowing:

- Halt execution when reaching a certain line
- Step through the program one instruction at a time
- Inspect values of variables after crashes
- Conditionally halt execution when a condition is met
- Many more advanced features

Should be covered in CMSC132!

Debugger Concepts

- Breakpoints: Pause execution at specific lines
- Step Over: Execute current line including function calls
- Step Into: Enter function to debug internals
- Step Out: Continue until function returns
- Call Stack: Sequence of function calls to current point
- Variable Inspection: Examine and modify values
- Watch Expressions: Monitor specific values during execution
- Conditional Breakpoints: Only halt when condition is true

System Call Tracing

Monitor interactions between a program and the OS kernel.

- Tools: `strace` (Linux), `dtruss` (macOS)
- Useful for debugging programs where you cannot see source code
- Shows every system call made by a program with arguments and results

```
# On Linux
sudo strace -e lstat ls -l > /dev/null
# On macOS
sudo dtruss -t lstat64_extended ls -l > /dev/null
```

Network Packet Analysis

Diagnose network-related issues by examining traffic between systems.

- Tools: `tcpdump` (command line), `Wireshark` (GUI)
- Captures and filters network packets
- Helpful for debugging network performance, security vulnerabilities, or data transmission

Browser Developer Tools

Essential for web development debugging (Chrome DevTools, Firefox Developer Tools).

- Open DevTools (F12)
- Sources tab $\rightarrow$ set breakpoints
- Step through code
- Inspect variables in Scope panel
- Console for live evaluation
- Network tab for requests
- Headers, Preview, Response, Timing
- "Copy as cURL" to replay
- Preserve log across navigation
- Performance analysis

Common Use Cases

Use Case	How
API debugging	Check request data & response structure
Performance issues	Use timing column to find slow requests
Cookie/Storage	Application tab shows cookies, localStorage
JavaScript errors	Console shows errors with stack traces
Live editing	Modify CSS/JS to test fixes without redeploying

Beyond Functional Correctness

Even if your code behaves correctly, it might not be good enough if it takes all your CPU or memory.

- Algorithms classes teach big O notation but not how to find hot spots
- Premature optimization is the root of all evil
- Profilers help you understand which parts take most time/resources
- Focus optimization efforts where they matter most

Types of Timing

When measuring program execution, there are different types of time:

Type	Description
Real Time	Total time from start to finish (wall clock)
User Time	CPU time spent running your program's code
Sys Time	CPU time spent on system-level tasks (OS calls, I/O)

Understanding the Differences

Wall clock time can be misleading:

- Your computer may run other processes simultaneously
- Program may wait for external events (network, disk I/O)
- Real Time = User Time + Sys Time + Wait Time

Key Insight

If your program spends lots of time waiting for external resources:

- Improve by reducing wait times, not making code faster
- Send fewer requests to servers
- Cache information to avoid re-checking files on disk

CPU Profilers: Tracing vs Sampling

Tools to analyze and measure CPU time spent on different functions/code blocks.

- Track every function call your program makes
- Complete record of execution
- Higher overhead
- Precise call counts

- Periodically sample program's stack (e.g., every millisecond)
- Aggregate statistics of time usage
- Lower overhead
- Statistical approximation

Python Example: cProfile

Python's `cProfile` module profiles time per function call:

```
import cProfile
import re

def grep_python(pattern, filename):
    with open(filename, 'r') as f:
        for line in f:
            if re.search(pattern, line):
                print(line.strip())

# Profile the function
cProfile.run('grep_python("import", "script.py")')
```

Memory Profiling Overview

Track, measure, and analyze memory usage to detect inefficiencies and prevent memory issues like leaks.

Types of Memory Profiling

Type	Purpose
Heap Profiling	Track dynamic memory allocations/deallocations
Stack Profiling	Monitor memory in call stack (local variables)
Reference Counting	Track how many references exist to each object

Optimization Strategy

Avoid unnecessary allocations:

- Memory allocation takes time and resources
- Garbage collection cycles introduce overhead
- Minimize allocations by reusing memory
- Use memory pools or object pooling techniques
- Avoid repeated allocations/deallocations

Flame Graphs: Hierarchical Visualization

Visualizations showing how much time a program spends in different functions.

- Y-axis: Call stack depth
- X-axis: Time/sample count proportions (NOT passage of time)
- Each box = function in call stack
- Width = time spent
- Height = call stack depth
- Wide boxes at top = bottlenecks

- Colors are random (visual distinction)
- Interactive: click to zoom
- Look for wide boxes
- Easy to identify hot paths

Generating Flame Graphs with py-spy

```
# Install py-spy
pip install py-spy

# Profile and generate flame graph
py-spy record -o profile.svg --duration 30 -- python script.py

# Or attach to running process
py-spy record -o profile.svg --pid 12345
```

Creates an interactive SVG file you can open in your browser.

Example Interpretation

If you see a wide box labeled `json.loads`, your program spends significant time parsing JSON. Optimize by:

- Parsing JSON once and caching the result
- Using a faster JSON library like `orjson`
- Reducing the amount of JSON data being parsed

Key Takeaways

- Systematic approach: Observe, Hypothesize, Test
- Print debugging with logging
- Interactive debuggers for deep dives
- Specialized tools for system/network
- Browser DevTools for web

- Measure before optimizing
- CPU profilers (tracing vs sampling)
- Memory profilers for leaks
- Flame graphs for visualization
- Focus on bottlenecks, not hunches

Final Thought

"Premature optimization is the root of all evil." Donald Knuth
Use profilers to find real bottlenecks before optimizing.