CSCB09 Software Tools and Systems Programming
Review Session

Marcelo Ponce

Winter 2025

Department of Computer and Mathematical Sciences - UTSC

Today's class

Very high-level overview of the concepts covered in the course.

- Linux/Unix
- Operating Systems
- C Memory Model
- Files | Input/Output Operations
- Automation: make
- Processes
- Inter-Process Communication
- Signals
- Sockets
- Shell Scripting
- IO Multiplexing
- Threads

Today's class

Very high-level overview of the concepts covered in the course.

- Linux/Unix
- Operating Systems
- C Memory Model
- Files | Input/Output Operations
- Automation: make
- Processes
- Inter-Process Communication
- Signals
- Sockets
- Shell Scripting
- IO Multiplexing
- Threads

Disclaimer
We will revisit the concepts in an "arbitrary" (logical) order, i.e. not chronologically as they have been presented in the course neither highlighting relevance nor significance in the final but instead conceptually coherence among them.

Review

Linux/Unix

Unix

- files and directories
- permissions
- utilities/commands

Shell

- programming/scripting
- quoting
- wild cards: $\boxed{*}$ $\boxed{?}$
- files

Shell Concepts

- `stdin`, `stdout`, `stderr`
- I/O Redirection: >, >>, <, ...
- Processes Control: `ps`, `kill`, ...
- Job Control: `fg`, `bg`, `&`
- Pipes: |

Shell Scripting

quoting

- single quotes `' ... '` inhibit wildcard replacement, variable substitution and command substitution.
- double quotes `" ... "` inhibit wildcard replacement only
- back quotes `` ` ... ` `` cause command substitution

variables – environment and local

- `str1="string"`
- `str2="string"`
- `if test $str1 = $str2; then ... fi`

Shell Scripting

- *shebang*: `#!/bin/sh`, `#!/bin/bash`, ...
- `test -f filename` – test if a file exists
  Many others ...
- Command line arguments
  `$0` name of the script
  `$1` ... `$n` arguments
- `set` assings positional arguments to a list of words
- `read` reads from `stdin`
- `expr` math functions

Compilation

Compiler *vs* Interpreter

- *Compiler* translates whole program to *object code*. Produces the most highly optimized code.
- Interpreter translates one line of code at a time. Can quickly make changes and try things out.
- C – compiled.
- Java – compiled to byte code, then interpreted.
- shell – interpreted.

Compilation Stages

1. Pre-processor
2. Compiler
3. Assembler
4. Linker

Software Tools

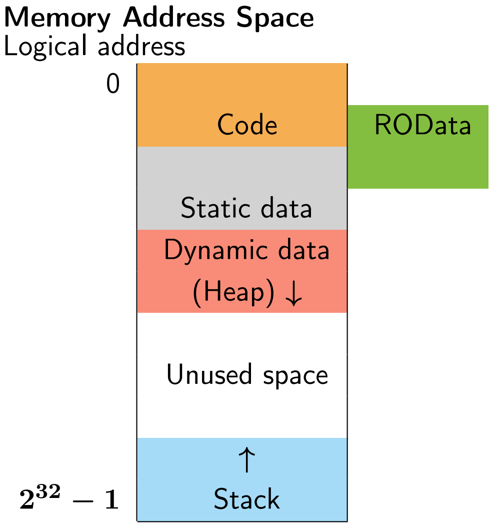
- Tools save you time and make you a better programmer:
  editor, language choice, debugger, build system, version control system, regression testing, issue tracking, profiling and monitoring.
- High-level scripting languages make it possible to glue programs together to do all kinds of time-saving tasks.

Programs as Data

- Executables are just files that can be copied, moved, searched and even edited.
- Compilers are just programs that operate on source code and produce executables.
- Programming tools treat program source code as data
- High-level programming languages give us easier ways to operate on programs: automated testing, build systems, version control

Programming in C

- Memory Model
  - pointers are addresses with a type.
- No variables are automatically initialized.
- Arrays
  - contiguous regions of memory with fixed size.
  - provide random access
- Pointers
  - *dereference* with $*$
  - get the address of a variable with &
- Functions
  - passing arguments *by-value* vs *by-reference*
  - arrays? – *decay*



String in C

- Remember the null termination character $\boxed{\texttt{'\textbackslash 0'}}$
- Most string functions depend on it.
- Whenever possible use the string functions rather than re-implementing them.
- E.g., use `strncpy` rather than copying each character.
- Be careful to ensure that you don't walk of the end of a character array.

Dynamic Memory Allocation

- `malloc`, `calloc`, `realloc`
- memory allocated using `malloc` should be freed when it is no longer needed
- keep a pointer to the beginning of the region so that it is possible to free
- **memory leak** occurs when the memory region is not freed

  E.g. you no longer have a pointer to a region of dynamically allocated memory
- **dangling pointers** – mistaken pointer-type

Dynamic Memory Allocation

- `malloc`, `calloc`, `realloc`
- memory allocated using `malloc` should be freed when it is no longer needed
- keep a pointer to the beginning of the region so that it is possible to free
- **memory leak** occurs when the memory region is not freed
  E.g. you no longer have a pointer to a region of dynamically allocated memory
- **dangling pointers** – mistaken pointer-type

When to use `malloc`

- When passing a pointer to a new region of memory back from a function
- When you don't know until runtime how much space you need
- When you are dealing with dynamically growing structures

Dynamic Memory Allocation

- `malloc`, `calloc`, `realloc`
- memory allocated using `malloc` should be freed when it is no longer needed
- keep a pointer to the beginning of the region so that it is possible to free
- **memory leak** occurs when the memory region is not freed

  E.g. you no longer have a pointer to a region of dynamically allocated memory
- **dangling pointers** – mistaken pointer-type

When to use `malloc`

- When passing a pointer to a new region of memory back from a function
- When you don't know until runtime how much space you need
- When you are dealing with dynamically growing structures
- This is a poor use of malloc:

```
char *str1 = malloc(MAXLEN);
```

Header Files

- Header files contain function prototypes and type definitions.
- Never `#include` a file containing functions and variable declarations file. You will run into trouble.
- Header files are useful when your program is divided into multiple files.
- Use *Makefiles* to compile programs. Saves typing and takes advantage of separate compilation.

`make`

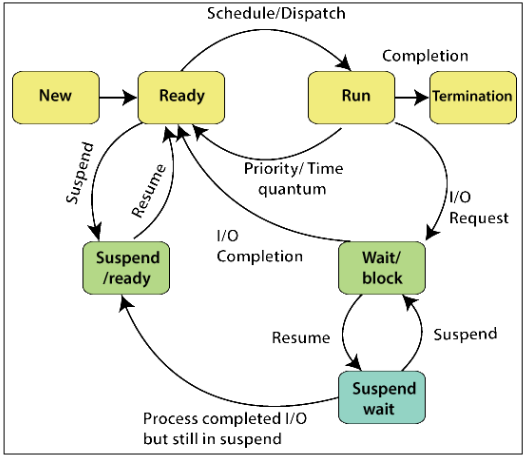
- Automation tool.
- Rules, Targets and Dependencies
- Macros and Variables

System Calls

- Perform a subroutine call into the Unix kernel
- Interface to the kernel
- main categories
  - file management
  - process management
  - error handling
  - communication
- Error handling
  - system calls usually return -1 (Always check!)
  - `errno`

Processes

- Process state:
  intialization, ready, running, blocked/waiting, suspended, terminated
- `fork()` – creates a duplicate process
- `exec()` – replaces the program being run by a different one
- *File Descriptors* maintained across `fork` and `exec`
- process ids: `getpid()`, `getppid()`



Process Termination

- Orphan Process:

  a process whose parent is the init process because its original parent died
- Zombie Process:

  a process that is "waiting" for its parent to accept its termination status.

  ```
  wait(int *status);
  ```

  ```
  r = waitpid(pid_t pid, int *status, int options);
  ```
- Use macros to check the status:

  `WIFEXITED`, `WIFSIGNALED`, `WEXITSTATUS`

Threads

- Processes have two limitations:
  - it is expensive to create a new one and switch between processes.
  - processes cannot share memory (easily)
- Threads allow multiple instruction streams (threads of execution) in a single address space and solve both these problems.
- Thread libraries also contain higher-level synchronization mechanisms (mutex's) and conditional variables.

Concurrency

- **Race condition**: final outcome depends on the order in which things run.
- Producer/Consumer Problem:
  - consumer should block when buffer is empty
  - producer should block when buffer is full
  - only one should be updating the buffer at a time
- A pipe is an example of producer/consumer

Inter-Process Communication (IPC)

- Data exchange between process:
  message passing: files (?), pipes, sockets
- Limitations of files for IPC data exchange
  - slow
  - possibly altered by other processes
- Limitations of pipes:
  - two processes must be running on the same machine
  - two processes must be related
- **Sockets** overcome these limitations

Inter-Process Communication (IPC)

- Data exchange between process:
  message passing: files (?), pipes, sockets
- Limitations of files for IPC data exchange
  - slow
  - possibly altered by other processes
- Limitations of pipes:
  - two processes must be running on the same machine
  - two processes must be related
- **Sockets** overcome these limitations
- Signals, semaphores, exit codes.

Streams / File Descriptors

- Unix has two main mechanisms for managing file access
  - **streams**: high-level, more abstract (and portable)
    - you deal with a pointer to a `FILE` structure, which keeps track of info you don't need to know
    - `fopen()`, `fprintf()`, `fread()`, `fgets()`
  - **file descriptors**: each file identified by a small integer (on Unix), low-level, used for files, sockets and pipes.
  - Binary versus text/ASCII I/O
- Files: *inodes*
- I/O Ops, FS, ...

Signals

- Signals are software interrupts, a way to handle asynchronous event.
- Examples: control-C, termination of child, floating point error, broken pipe.
- Normal processes can send signals.
- `kill(pid, SIG)` – sent `SIG` to `pid`
- `sigaction()` – install a new signal handler for a signal

Sockets i

- Sockets allow communication between machines/hosts/devices
- TCP/IP protocol – internet address, ports
- Protocol families: `PF_INET`, `PF_LOCAL`
- Server side initialization takes 4 steps
  - `socket()` – initialize protocol
  - `bind()` – initialize addresses
  - `listen()` – initialize kernel structures for pending connections
  - `accept()` – block until a connection is received

Sockets ii

- Client initializes socket using `socket()`, and then calls `connect()`
- Need to be wary of host byte orders.
- Communication is done by reading and writing on file descriptors.
- Ports are divided into three categories: well-known, registered, and dynamic (or private).
- Socket types:

```
SOCK_STREAM = TCP
SOCK_DGRAM = UDP
```

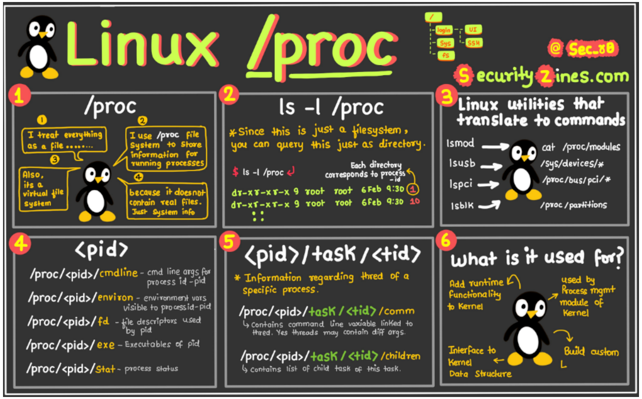
Multiplexing I/O

- `select()` allows a process to block on a set of file descriptors until one or more of them are ready.
- Read calls on a "ready" file descriptor will only block while the data is transferred from kernel to user space.
- Makes it easier for one process to handle multiple sources of input.
- `select()` takes "file descriptor sets" as arguments
- The macros `FD_SET`, `FD_ISSET`, etc. are used to manipulate the bit set data structure.

File Interface

- **"Everything is a file"**
- We treat all sorts of *devices* as if they were files, and use the file interface (open, read, write, close) all over the place.
  - files
  - directories
  - pipes
  - sockets
  - kernel info via `/proc`

* `/proc` virtual sub-file system



1 /proc

1 I treat everything as a file.......

2 I use /proc file system to store information for running processes

3 Also, its a virtual file system

4 because it doesnot contain real files. Just system info

2 ls -l /proc

*Since this is just a filesystem, you can query this just as directory.

```
$ ls -l /proc
dr-xr-xr-x 9 root root 6Feb 9:30 1
dr-xr-xr-x 9 root root 6Feb 9:30 10
:
```

Each directory corresponds to process-id

3 Linux utilities that translate to commands

lsmod	cat /proc/modules
lsusb	/sys/devices/*
lspci	/proc/bus/pci/*
lsblk	/proc/partitions

4 <pid>

/proc/<pid>/cmdline - cmd line args for process id - pid

/proc/<pid>/environ - environment vars visible to process id-pid

/proc/<pid>/fd - file descriptors used by pid

/proc/<pid>/exe - Executables of pid

/proc/<pid>/Stat - process status

5 <pid>/task/<tid>

* Information regarding thred of a specific process.

/proc/<pid>/task/<tid>/comm
↳ Contains command line variable linked to thred. Yes threads may contain diff args.

/proc/<pid>/task/<tid>/children
↳ contains list of child task of this task.

6 What is it used for?

Add runtime Functionality to Kernel

used by Process mgmt module of Kernel

Interface to Kernel Data Structure

Build custom

Unix Philosophy

- Write programs that do one thing well.
- Write programs that work together.
- Write programs to handle text streams because that is the universal interface.

Final Exam

Final Exam

- Course Material & How to study
  - Classes
  - Textbook
  - Lecture slides
  - PCRS
  - Tutorials: workout examples, worksheets and labs
  - Assignments – make sure you understand concepts and code
- Covers everything in the course
- Closed book exam – No Aids Allowed

- Date: Thursday April 17th, 2025
- Time: 9:00-12:00
- Location: IC-130
- Aids: none.

The End

That's it...

Good luck with your final exams!
Study hard!
See you on Thursday!