# Project 4: Shell Project: After the bomb, shell shock!

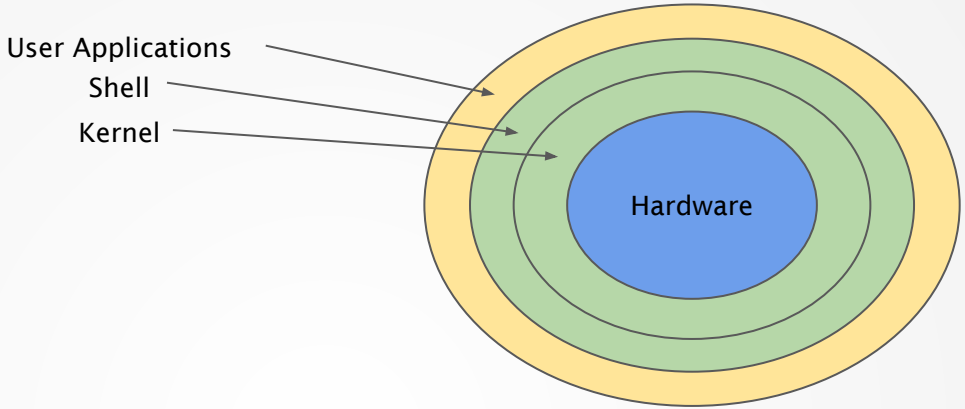
## CS 0449: Introduction to System Software

CS0449 Teaching Assistants

University of Pittsburgh | School of Computing and Information

# Project 4

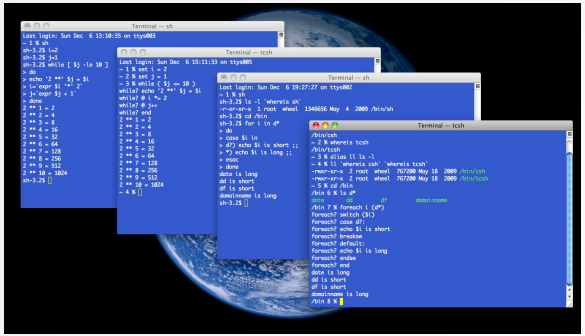
Writing your own shell




# The shell

is the outermost layer of the operating system

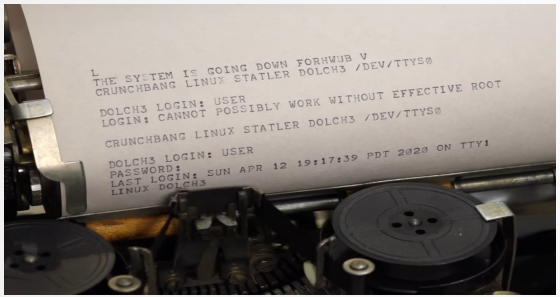
# What's a *shell*?

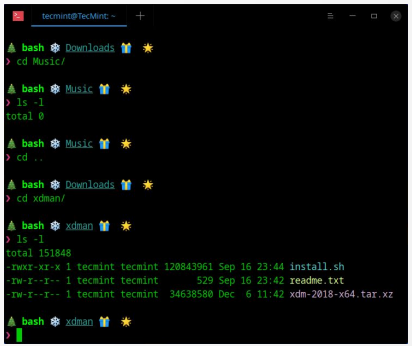


- It's the *"command line"*
- A ***shell*** is an application program that runs programs on behalf of the user.
- Typically a shell is a program that
  1. Repeatedly prints a prompt
  2. Waits for a *command line* on `stdin`
  3. Carries out some action (as directed by the contents of the command line)
- A ***Read*** → ***Evaluate*** → ***Print*** loop *(REPL)*

# Some terminology

- A **shell** is a user interface for accessing an computer system
- Most often the user interacts with the shell using a **command-line interface** (**CLI**).
- The **terminal** is a program that opens a graphical window and lets you interact with the shell.
  - Actually this is a **terminal emulator** or **virtual console**
  - Technically, terminals are physical machines that provides an interface with a larger machine
    - Teletypewriters
    - Video display terminals
- In reality, all these terms are *more or less* used interchangeably.



# Many different shells, including your very own!

- There are various different shells that you can use.
  - `sh` – Original Unix shell (Stephen Bourne, AT&T Bell Labs, 1977)
- Most common is the Bourne-Again shell (bash)
  - Installed with most Linux distributions
  - Just another program → `/bin/bash`
- Some others include:
  - Z-shell (zsh) → `/bin/zsh`
    - Preinstalled for modern MacOS, modern Linux distributions
  - PowerShell, COMMAND.COM
    - For Windows
    - Not a Unix-Shell
  - fish/csh, and much more
- For project IV, you will implement your very own shell
  - Primitive, yet still functional
  - *It accomplishes all that needs to be done*

# msh specification *Hopefully you can come up with a good name for your shell that ends with "-sh"*

## Your shell should:

- Print a prompt: "> "
- Read user input
  - The command line input by the user consists of a *name* and zero or more arguments (delimited by spaces)

```
> ls              # command: ls; arguments: ls
> ls -a           # command: ls; arguments: ls, -a
> exit            # command: exit; arguments: exit
> load better_ls # command: load; arguments load, better_ls
```

# msh specification

## Your shell should:

- Support built-in commands
  - `exit`: The shell should exit upon receiving this command
  - `load`: The shell should dynamically load a plugin and initialize it
- Support extensioning built-in commands via plugins
  - Plugin Interface:
    - `int initialize()`
      - Returns `0` on success
    - `int run(char **argv)`
      - `argv`: array of Strings terminated by `NULL`
        - `argv = {"ls", "-a", NULL}`
      - Returns `0` on success
  - Throw error message if plugin could not be loaded
    `Error: Plugin <plugin> initialization failed!`
  - Once loaded, user should be able to run the extended functionality by invoking the plugin's name

# msh specification

## Your shell should:

- Support extensioning built-in commands via plugins

```
> broken_better_ls   # Not loaded
> load broken_better_ls
Error: Plugin broken_better_ls initialization failed!
> broken_better_ls   # Still not loaded
> better_ls          # Not loaded
> load better_ls     # Success
> better_ls          # Loaded
msh    msh.c     better_ls.c   better_ls.so
>
```

# msh specification

Your shell should:

- Allow for instantiating other executables *and pass in arguments*

```
shk148@thoth $ ./msh
> vim better_ls.c
> gcc better_ls.c -o better_ls.so -shared
> load better_ls
> better_ls
msh     msh.c      better_ls.c    better_ls.so
> exit
shk148@thoth $
```

# msh specification limitations

To simplify your implementation, testing will be limited to:

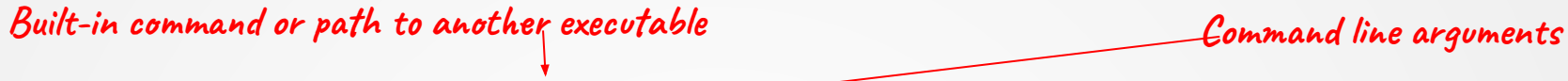
1. Commands will have a maximum size of 200 characters
2. Program names and arguments will have a maximum size of 20 characters
3. There will be at most 20 arguments
4. Your shell need only support loading upto 10 plugins

# Building the shell: ~~Skeleton~~ Shelleton

```
int main(){
    while (TRUE)   When do we break out of this loop?
    {    /* Infinite Loop for REPL */
         PrintCommandPrompt()
         cmdLine = readFromStdIn();
         cmd = parseCommand(cmdLine);
         If (cmd is BuiltInCommand) {executeBuiltInCommand(cmd)};
         Else
         {   If the command not a built-in command, we should check if it's a name of an executable file
              fork()
              // Child process should run the executable
              What should the parent process do while the child process is running?
         }
    }
}
```

...This is just one approach to building your shell

# Building the shell: Reading and Parsing Input




- A command goes in → 📦 → a process comes out
  - A shell, at its simplest, is a program that reads input from the user and tries to execute commands.
- We can read in a line of input using `fgets()`
- Given a user input, we need to categorize it as
  - Built in command or
  - Name of an executable
- But before we can interpret the input, we need to tokenize it

```
"ls -l -a /usr" /* delimited by ' '*/
    ⇒ {"ls", "-l", "-a", "/usr"}
```

# `man strtok` abridged

- The `strtok()` function can help tokenize strings
- `#include <string.h>`
- `char *strtok(char *str, const char *delim);`
  - Breaks string `str` into a series of tokens using the delimiter `delim`.
  - Returns a pointer to the next token, or NULL if there are no more tokens.
- Called in one of two ways:

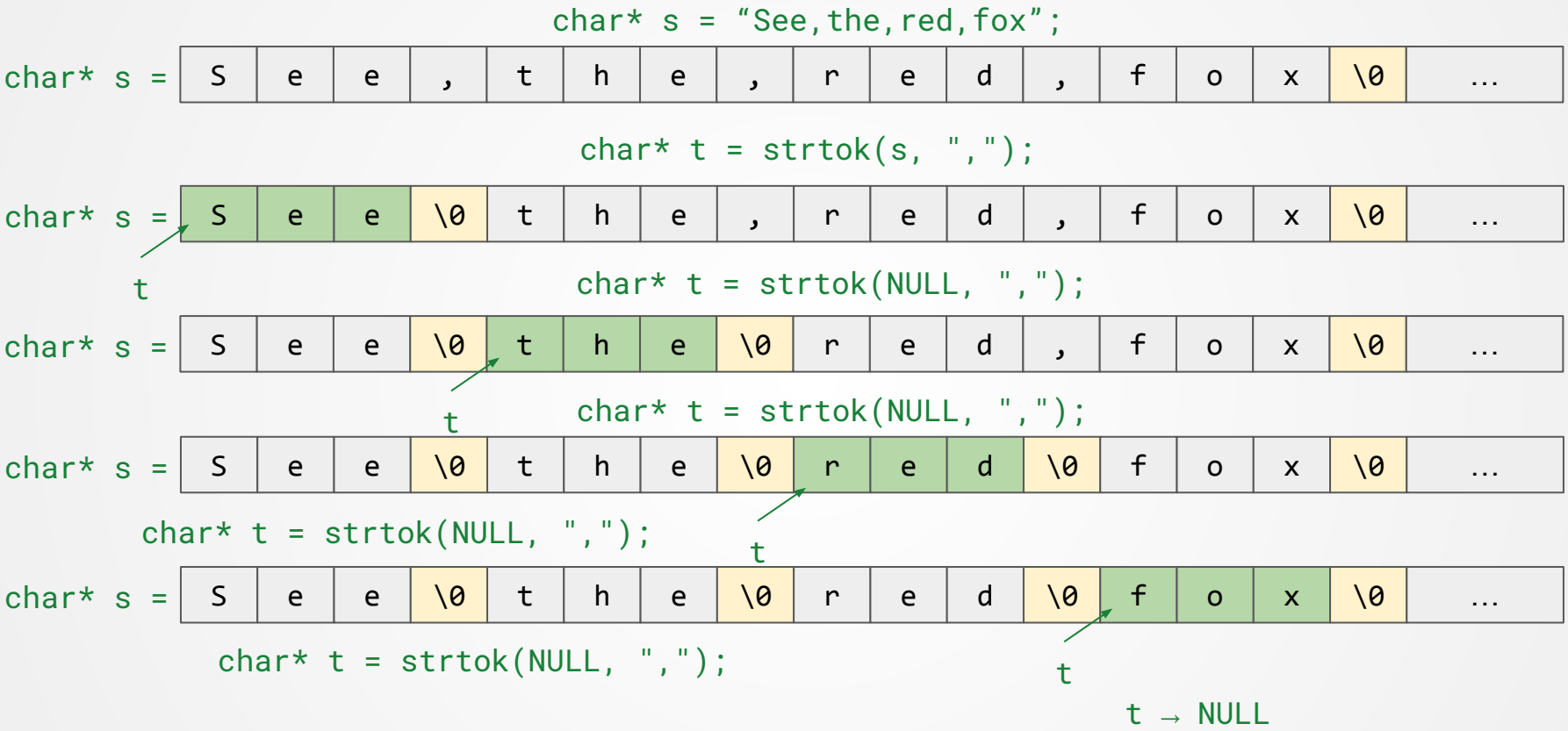
1. `strtok(str, d) // starts processing a new string`
2. `strtok(NULL, d) // continue processing a string`

# A `strtok()` example

```c
#include <stdio.h>
#include <string.h>

int main() {
    char str[] = "apple,orange;banana-grape";
    char delimiters[] = ",;-";

    // Using strtok to tokenize the string
    char *token = strtok(str, delimiters);

    // Loop through the tokens and print them
    while (token != NULL) {
        printf("Token: %s\n", token);

        // Get the next token
        token = strtok(NULL, delimiters);
    }

    return 0;
}
```

```
$ ./strtok_example

Token: apple
Token: orange
Token: banana
Token: grape
```

# A `strtok()` example




- `strtok()` changes the string that has been parsed!

# `man strtok` #NOTES-AND-BUGS

- Be cautious when using these functions. If you do use them, note that:
  - These functions modify their first argument.
  - These functions cannot be used on constant strings.
  - The identity of the delimiting byte is lost.
- For instance, if you try
  - `strtok("String Constant", delim)`
  - Segmentation fault! (attempting to write to a literal)

# Still unsure? Read the man pages!

```
$ man strtok
```

- What arguments does the function take?
  - read **SYNOPSIS**
- What does the function do?
  - read **DESCRIPTION**
- What does the function return?
  - read **RETURN VALUES**
- What errors can the function fail with?
  - read **ERRORS**
- Is there anything I should watch out for?
  - read **NOTES**
- I want an example
  - read **EXAMPLES**
  - https://pitt.edu/~shk148/teaching/CS0449-2234/code/strtok.c.html

# `strtok()` vs `strsep()`

- Alternatively, you can use `strsep()`
  - `#include <string.h>`
  - `char *strsep(char **stringp, const char *delim);`
- A *replacement* for `strtok()`
- But not all C versions support it
  - For instance, ANSI-C does not support `strtok()`
  - Hence, it is *less portable*
- You may use either `strsep()` or `strtok()` in this project
  - Read the documentation (`man strsep`) to see how each work!

# Building the shell: Executing command

- Once we've tokenized the input, we can use standard C-string functions to *compare*
  - `strcmp()` and friends
- If the keyword matches a built-in command
  - Run it!
  - Some functionalities may require dynamically loading *plugins*
    - Just as you did for lab 6
- If the keyword is unknown,
  - It's probably the name of an executable
  - So run it!
    - `fork()` and friends
      - `exec*()`
        - `wait()`

# Building the shell: Executing command

- Once we've tokenized the input, we can use standard C-string functions to *compare*
  - `strcmp()` and friends
- If the keyword matches a built-in command
  1. `exit` ⇒ Exit the program
  2. `load` ⇒ Dynamically load plugins (just like lab 6)
     - Since our shell needs to support dynamically loading multiple plugins
       - Devise some data structure to store them
       - Create helper functions to add and access plugins

# Building the shell: Executing command

- If the keyword does not match a built-in command
- Check if it's a plugin
  - and run it
- If it's not a plugin
  - It must be an executable name
  - `fork()`, `exec*()`, and their friends!
    - Make sure to use the correct `exec*()` function
    - And correctly pass in arguments

# Implementation Hints

1. When multiprogramming with `fork()`s
   - Think about the order in which processes need to run
   - Does a process need to wait for another?
2. String parsing is weird and hard
   - Especially since the standard functions exhibits odd behavior
   - Carefully read the documentation
   - Verify output before moving onto next step
3. There is a lot to program
   - Break your program down into smaller functions
   - `readInput()`, `parseInput()`, `runBuiltIn()`, …
   - To pass values between functions, you have to store them in the heap!

➢ Since this project requires access to many standard library functions, we highly recommend developing on Thoth or another Linux machine

- And plan for outages!
  - ➢ Back-up frequently (to your local machine)

# Implementation Challenges

1. This project ties in everything you've learned so far
   - C programming & debugging
     - ➢ *See Lab0 (Hello lab)*
   - C-Strings and standard library functions
     - ➢ *See Project I (BMP Steganography) for a guide*
   - Maintaining data structures in C
     - ➢ *Lab4 (Queue lab)*
   - Pointers and management of memory
     - ➢ *See Lab3 (Pointer lab), Project II (Malloc)*
   - Process management and dynamic loading
     - ➢ *See Lab6 (Loading and Forking)*
2. One common issue: *Memory leaks*
   - Not maintaining pointers
   - `malloc()` without `free()`
   - Test your code for memory leaks using **valgrind**!

# Implementation Challenges

## 2. One common issue: *Memory leaks*

```
$ valgrind --leak-check=full --show-leak-kinds=all ./msh
```

```
HEAP SUMMARY:
==630754==     in use at exit: 3,683 bytes in 6 blocks
==630754==   total heap usage: 8 allocs, 2 frees, 5,731 bytes allocated
==630754==
==630754== 820 (808 direct, 12 indirect) bytes in 1 blocks are definitely lost in loss record 4 of 5
==630754==    at 0x484DA83: calloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==630754==    by 0x10981F: get_user_input (luis.c:134)
==630754==    by 0x1097D7: main (luis.c:124)
==630754==
==630754== 2,050 bytes in 2 blocks are definitely lost in loss record 5 of 5
==630754==    at 0x484DA83: calloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==630754==    by 0x10983A: get_user_input (luis.c:137)
==630754==    by 0x1097D7: main (luis.c:124)
==630754==
==630754== LEAK SUMMARY:
==630754==    definitely lost: 2,858 bytes in 3 blocks
==630754==    indirectly lost: 12 bytes in 1 blocks
==630754==      possibly lost: 0 bytes in 0 blocks
==630754==    still reachable: 813 bytes in 2 blocks
==630754==         suppressed: 0 bytes in 0 blocks
```

# Debugging

- Debugging this project is hard
  - So many functionalities to look out for
    - So many places to go wrong
    - So many places to shoot yourself in the foot
  - Measure twice, cut once!
- This project is fairly open-ended in its implementation
  - *You should be able to explain your own code!*
  - *"I wrote it and it sort of works, but I don't know why" ←BAD!*

# Works Referred

- Creative Commons photography courtesy of Arnold Reinhold and technikum29 via the Wikimedia Foundations
- `strtok()` examples adapted from Weber State University