Project 4: Writing your own shell

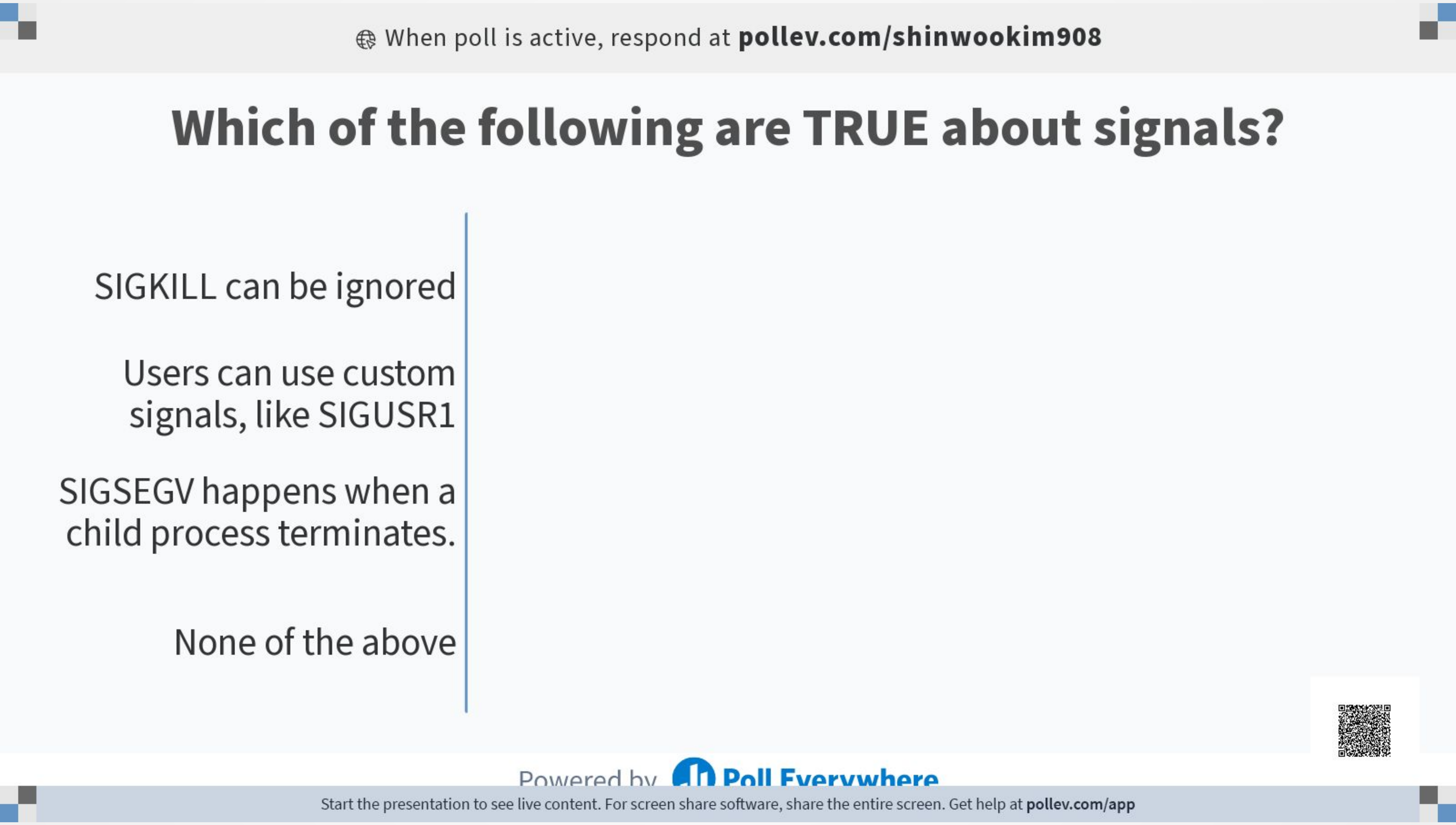
Shinwoo Kim
Teaching Assistant
shinwookim@pitt.edu
https://www.pitt.edu/~shk148/

Spring 2023, Term 2234
Friday 12 PM Recitation
5502 Sennott Square
Mar 2nd, 2023

Course News!

- Exams
  - Exam I grades were returned on March 24th, 2023
    - Check your email for class statistics
    - Request regrades if needed (this may adjust your grade up or down)
  - Exam II was on March 30th, 2023 during lecture
    - Still a few people who haven't taken it yet…so won't discuss
- Labs
  - Lab 5 (Process Lab) was due on March 30th, 2023 @ 11:59 PM EST
- Projects
  - Project III: Late submission closed on March 27th, 2023 @11:59 PM EST
    - Remember to schedule check-off meetings if you haven't already
  - Project IV was released on March 30th, 2023
    - **Due: April 10th, 2023 @ 11:59 PM EST**
- Poll Everywhere
  - www.pollev.com/shinwookim908
  - Solutions to recitation questions will be posted on website

PEV: Signals



When poll is active, respond at **pollev.com/shinwookim908**

Which of the following are TRUE about signals?

SIGKILL can be ignored

Users can use custom signals, like SIGUSR1

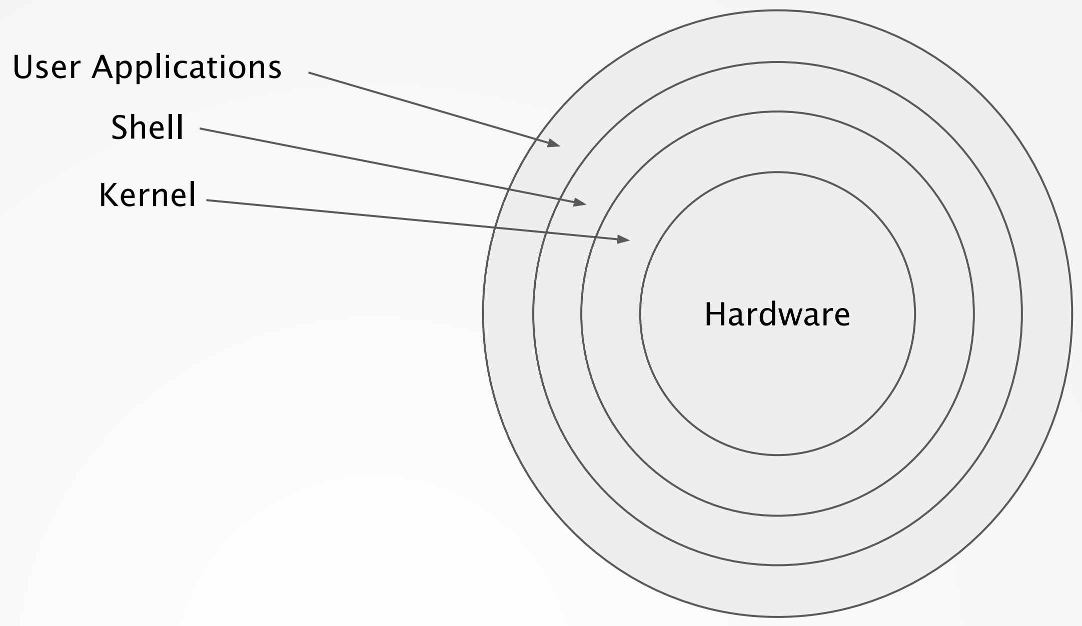
SIGSEGV happens when a child process terminates.

None of the above

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Project IV

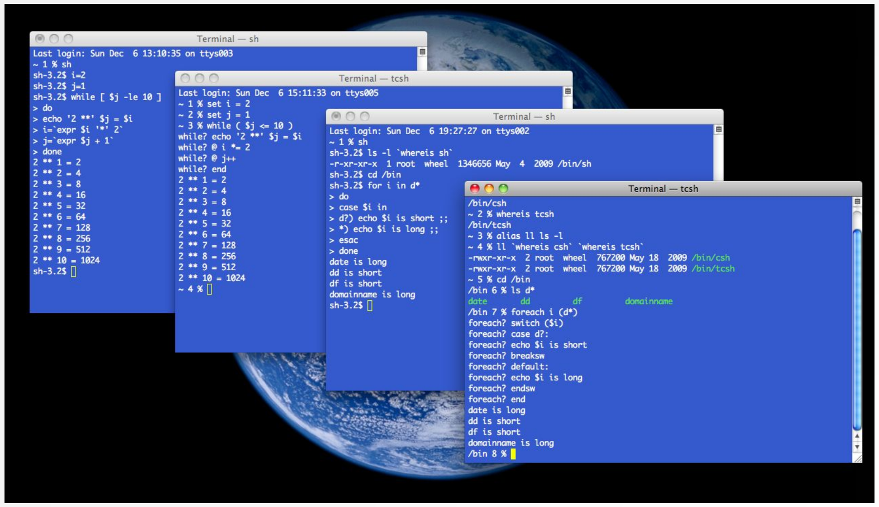
Writing your own shell



The shell

is the outermost layer of the operating system

What’s a *shell*?



```
Last login: Sun Dec  6 13:10:35 on ttys003
~ 1 % sh
sh-3.2$ i=2
sh-3.2$ j=1
sh-3.2$ while [ $j -le 10 ]
> do
> echo '2 **' $j = $i
> i=`expr $i '*' 2`
> j=`expr $j + 1`
> done
2 ** 1 = 2
2 ** 2 = 4
2 ** 3 = 8
2 ** 4 = 16
2 ** 5 = 32
2 ** 6 = 64
2 ** 7 = 128
2 ** 8 = 256
2 ** 9 = 512
2 ** 10 = 1024
sh-3.2$
```

```
Last login: Sun Dec  6 15:11:33 on ttys005
~ 1 % set i = 2
~ 2 % set j = 1
~ 3 % while ( $j <= 10 )
while? echo '2 **' $j = $i
while? @ i *= 2
while? @ j++
while? end
2 ** 1 = 2
2 ** 2 = 4
2 ** 3 = 8
2 ** 4 = 16
2 ** 5 = 32
2 ** 6 = 64
2 ** 7 = 128
2 ** 8 = 256
2 ** 9 = 512
2 ** 10 = 1024
~ 4 %
```

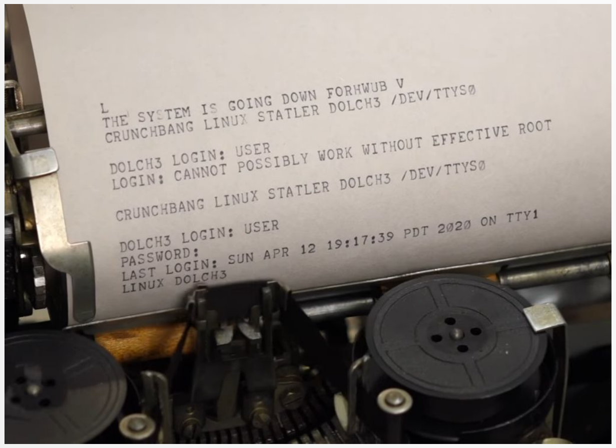
```
Last login: Sun Dec  6 19:27:27 on ttys002
~ 1 % sh
sh-3.2$ ls -l `whereis sh`
-r-xr-xr-x  1 root  wheel  1346656 May  4  2009 /bin/sh
sh-3.2$ cd /bin
sh-3.2$ for i in d*
> do
> case $i in
> d?) echo $i is short ;;
> *) echo $i is long ;;
> esac
> done
date is long
dd is short
df is short
domainname is long
sh-3.2$
```

```
/bin/csh
~ 2 % whereis tcsh
/bin/tcsh
~ 3 % alias ll ls -l
~ 4 % ll `whereis csh` `whereis tcsh`
-rwxr-xr-x  2 root  wheel  767200 May 18  2009 /bin/csh
-rwxr-xr-x  2 root  wheel  767200 May 18  2009 /bin/tcsh
~ 5 % cd /bin
/bin 6 % ls d*
date       dd         df         domainname
/bin 7 % foreach i (d*)
foreach? switch ($i)
foreach? case d?:
foreach? echo $i is short
foreach? breaksw
foreach? default:
foreach? echo $i is long
foreach? endsw
foreach? end
date is long
dd is short
df is short
domainname is long
/bin 8 %
```

- 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.

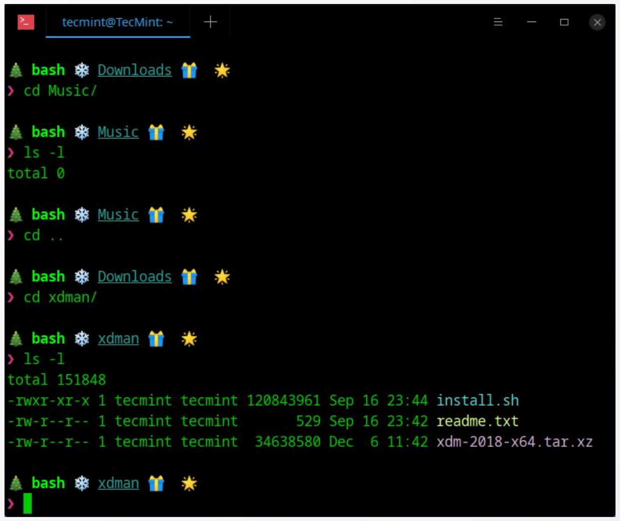


```
L
THE SYSTEM IS GOING DOWN FORHWUB V
CRUNCHBANG LINUX STATLER DOLCH3 /DEV/TTYSØ

DOLCH3 LOGIN: USER
LOGIN: CANNOT POSSIBLY WORK WITHOUT EFFECTIVE ROOT

CRUNCHBANG LINUX STATLER DOLCH3 /DEV/TTYSØ

DOLCH3 LOGIN: USER
PASSWORD:
LAST LOGIN: SUN APR 12 19:17:39 PDT 2Ø2Ø ON TTY1
LINUX DOLCH3
```



```
🎄 bash ❄ Downloads 🎁  🌟
❯ cd Music/

🎄 bash ❄ Music 🎁  🌟
❯ ls -l
total 0

🎄 bash ❄ Music 🎁  🌟
❯ cd ..

🎄 bash ❄ Downloads 🎁  🌟
❯ cd xdman/

🎄 bash ❄ xdman 🎁  🌟
❯ ls -l
total 151848
-rwxr-xr-x 1 tecmint tecmint 120843961 Sep 16 23:44 install.sh
-rw-r--r-- 1 tecmint tecmint       529 Sep 16 23:42 readme.txt
-rw-r--r-- 1 tecmint tecmint  34638580 Dec  6 11:42 xdm-2018-x64.tar.xz

🎄 bash ❄ xdman 🎁  🌟
❯ 
```

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)
  - Preinstalled with most Linux distributions
    - It's the one that's installed on Thoth
  - Just another program → `/bin/bash`
- Some others include:
  - Z-shell (zsh) → `/bin/zsh`
    - Comes 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:

- Instantiating other executables

```
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

Your shell should:

- Instantiating other executables

```
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?
            }
    }
}
```

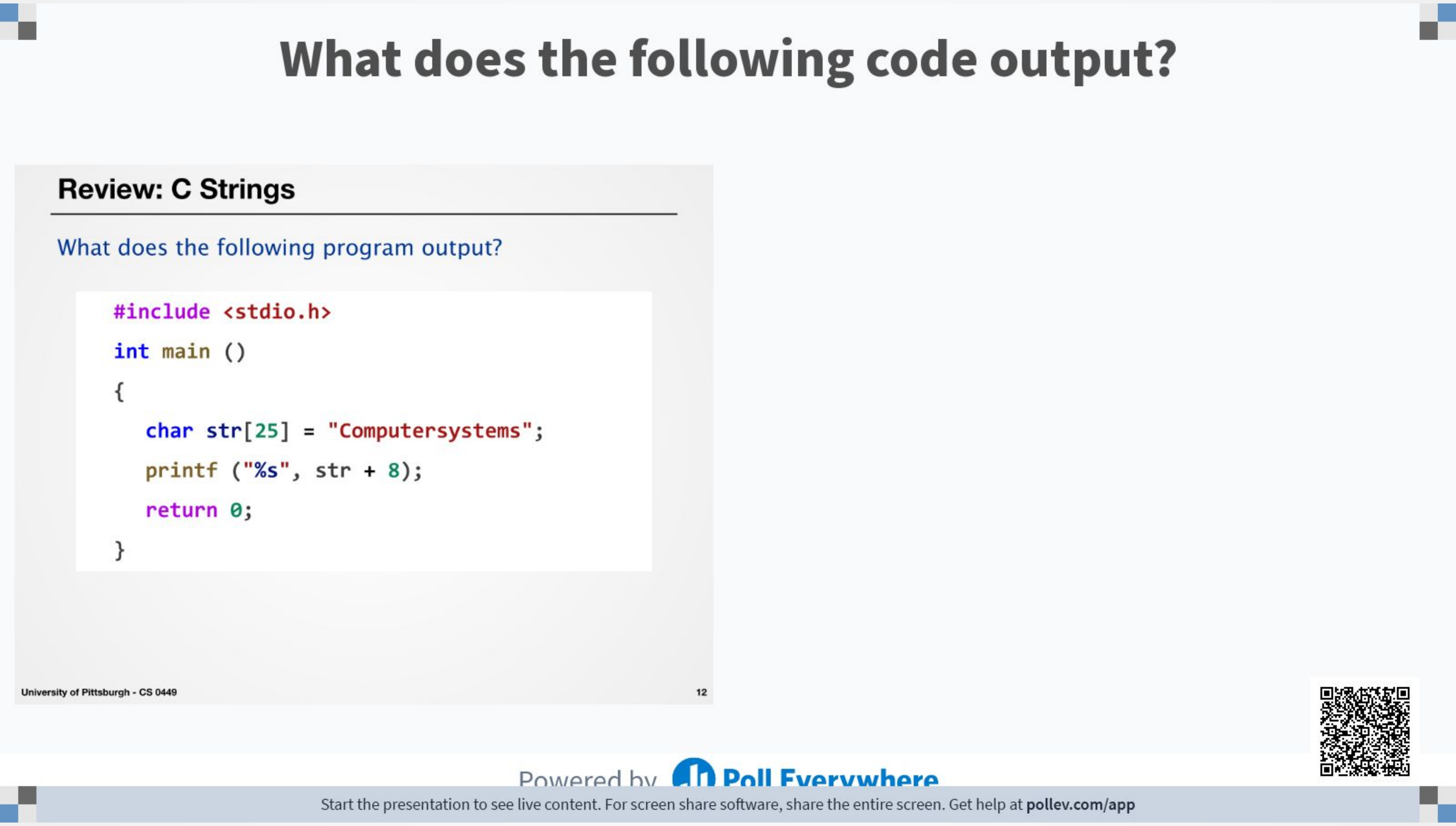
...This is just one approach to building your shell

Review: C Strings

What does the following program output?

```
#include <stdio.h>
int main ()
{
    char str[25] = "Computersystems";
    printf ("%s", str + 8);
    return 0;
}
```

PEV: C Strings



What does the following code output?

Review: C Strings

What does the following program output?

```
#include <stdio.h>
int main ()
{
   char str[25] = "Computersystems";
   printf ("%s", str + 8);
   return 0;
}
```

University of Pittsburgh - CS 0449 12

Powered by Poll Everywhere

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Building the shell: Reading and Parsing Input

Built-in command or path to another executable

Command line arguments

```
$ ls -l -a /usr
```

- 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 `scanf()`
- 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"}
```

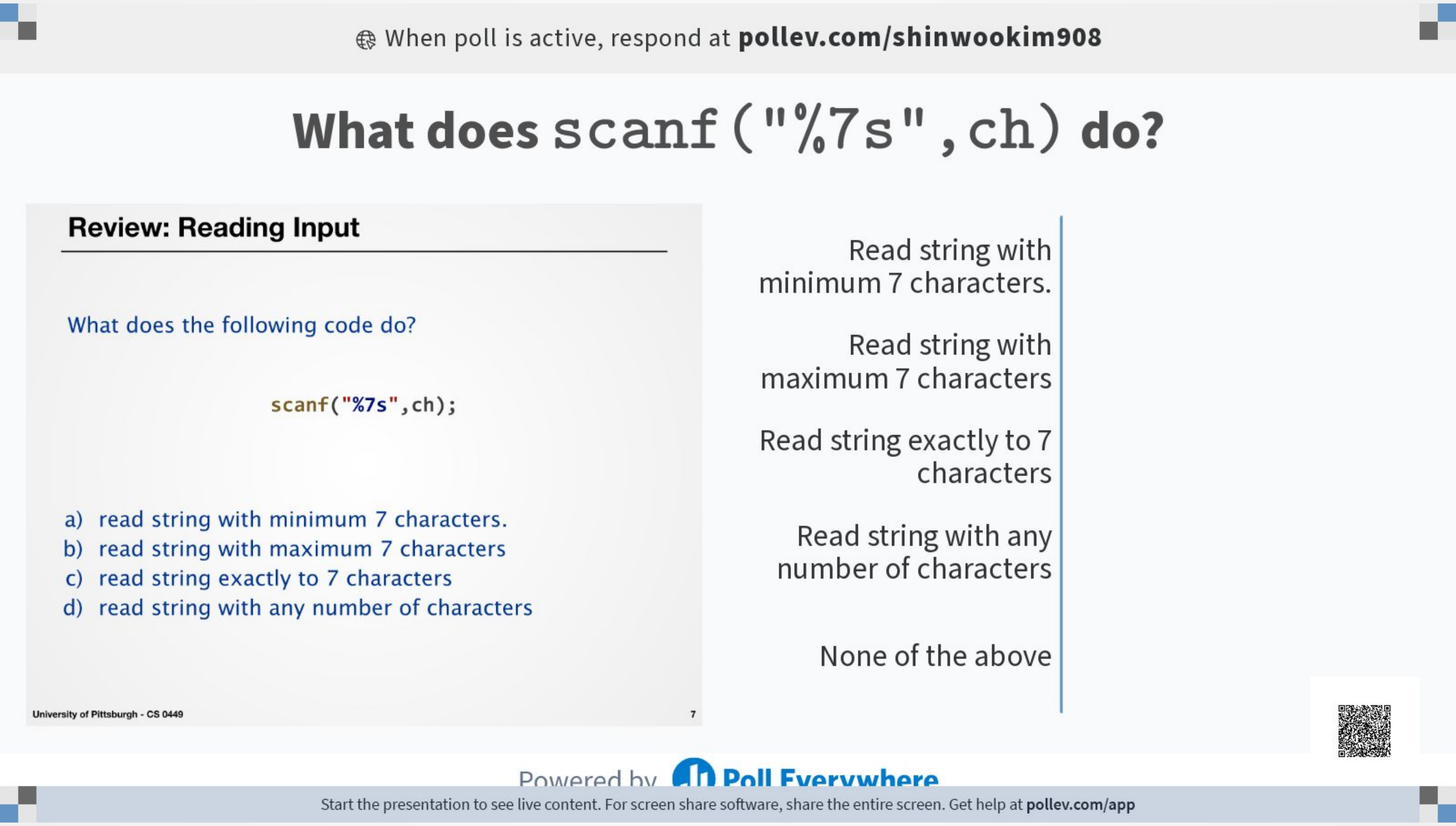
Review: Reading Input

What does the following code do?

```
scanf("%7s",ch);
```

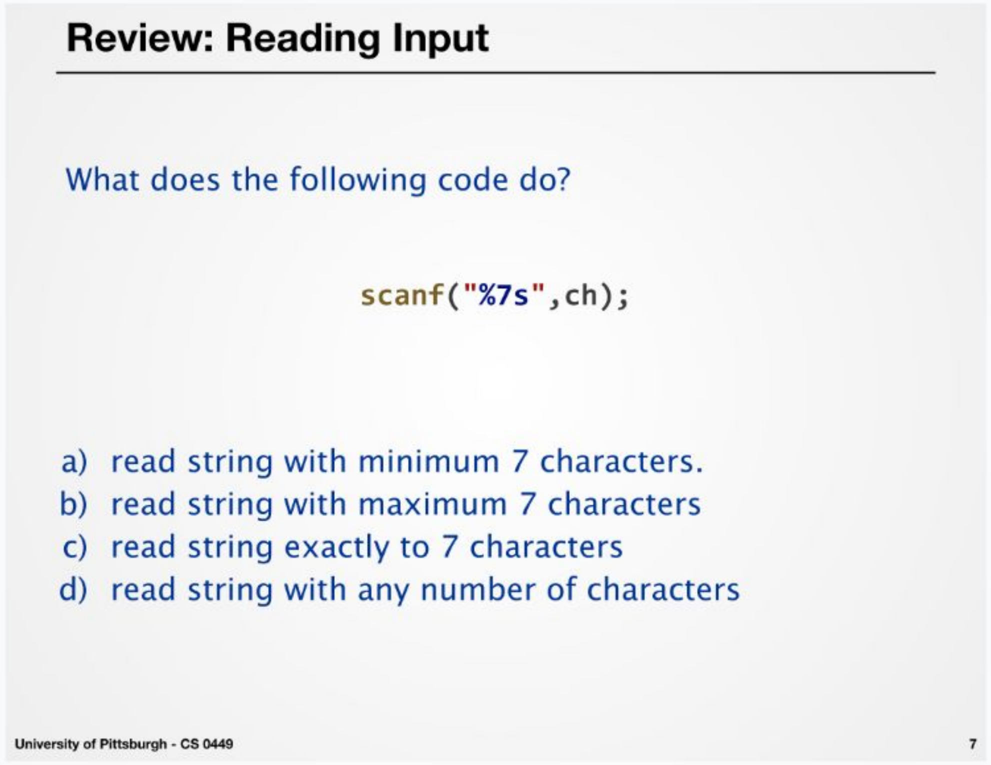
a) read string with minimum 7 characters.
b) read string with maximum 7 characters
c) read string exactly to 7 characters
d) read string with any number of characters

PEV: Reading Input



When poll is active, respond at **pollev.com/shinwookim908**

What does `scanf("%7s",ch)` do?



Review: Reading Input

What does the following code do?

```
scanf("%7s",ch);
```

a) read string with minimum 7 characters.
b) read string with maximum 7 characters
c) read string exactly to 7 characters
d) read string with any number of characters

University of Pittsburgh - CS 0449 7

Read string with minimum 7 characters.

Read string with maximum 7 characters

Read string exactly to 7 characters

Read string with any number of characters

None of the above

Powered by Poll Everywhere

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

`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

```
$ ./strtok_example

I
```

```c
#include <stdio.h>
#include <string.h>
int main(){
    char str[] = "I:love-programming";
    char delim[] = "-:";
    char *token;
    token = strtok(str, delim);
    printf("%s\n", token);
    return 0;
}
```

What will be printed?

A `strtok()` example

```
$ ./strtok_example

I
I
```

🤔 *But the second token should be "love"*

```c
#include <stdio.h>
#include <string.h>
int main(){
   char str[] = "I:love-programming";
   char delim[] = "-:";
   char *token;
   token = strtok(str, delim);
   printf("%s\n", token);
   token = strtok(str, delim);
   printf("%s\n", token);
   return 0;
}
```

What will be printed?

A strtok() example

```
$ ./strtok_example

I
love
```

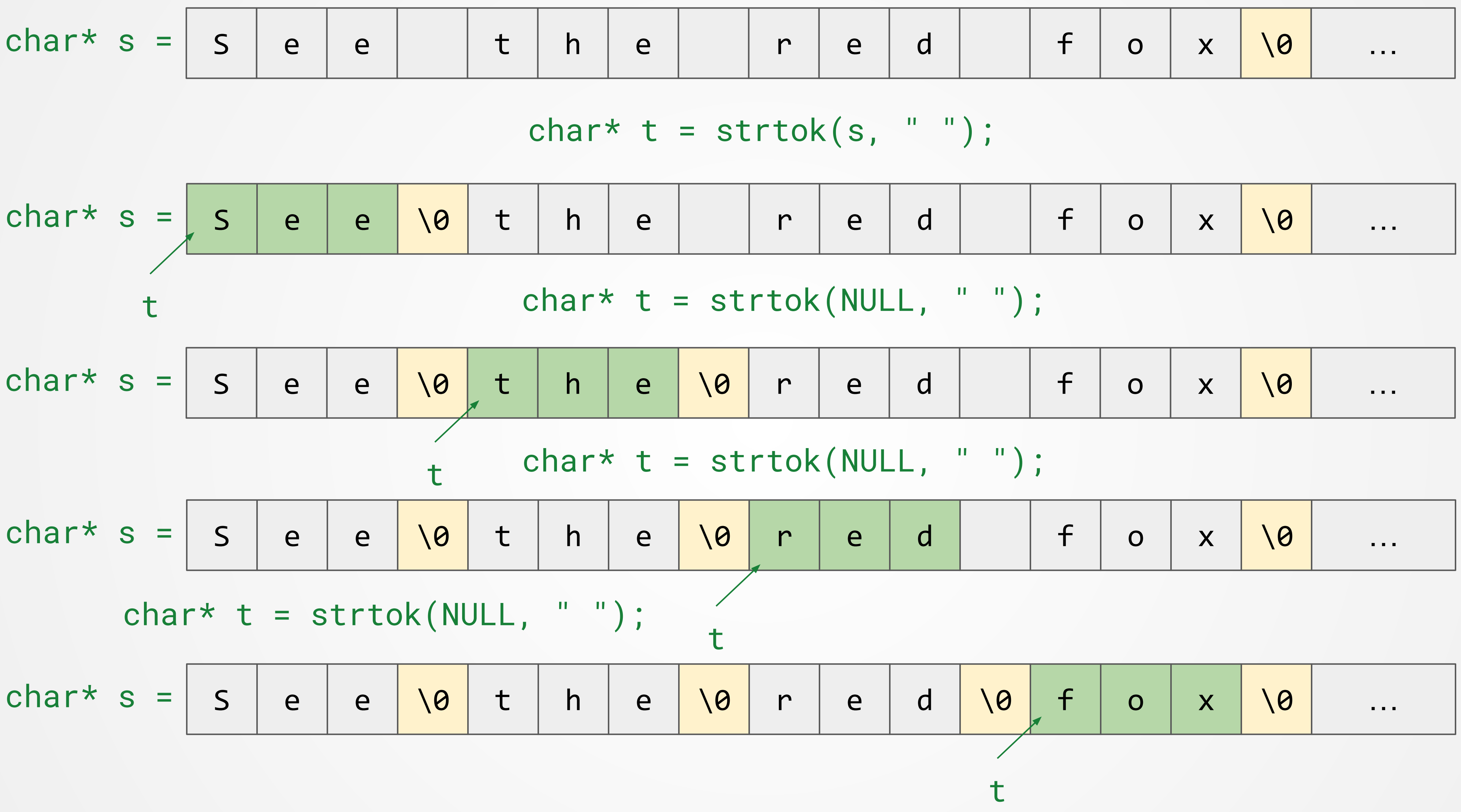
How can we print the remaining tokens?

```c
#include <stdio.h>
#include <string.h>
int main(){
   char str[] = "I:love-programming";
   char delim[] = "-:";
   char *token;
   token = strtok(str, delim);
   printf("%s\n", token);
   token = strtok(NULL, delim);
   printf("%s\n", token);
   return 0;
}
```

What will be printed?

A `strtok()` example

`char* s = “See the red fox”;`


char* s =
S e e t h e r e d f o x \0 ...
char* t = strtok(s, " ");
char* s =
S e e \0 t h e r e d f o x \0 ...
t
char* t = strtok(NULL, " ");
char* s =
S e e \0 t h e \0 r e d f o x \0 ...
t
char* t = strtok(NULL, " ");
char* s =
S e e \0 t h e \0 r e d f o x \0 ...
char* t = strtok(NULL, " ");
t
char* s =
S e e \0 t h e \0 r e d \0 f o x \0 ...
t

`char* t = strtok(NULL, " ");`   `t → NULL`

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

idem·po·tent

- The `strtok()` function exhibits some weird behavior
  - `strtok()` changes the string that has been parsed
  - Replacing the character in place with a null terminator (`'\0'`)
- `strtok()` produces different results when called multiple times
  - It’s a **non-idempotent** function
    - Which has ***side effects***.
- In comparison, functions that have no side effects are called **idempotent**.

```
x = 2; // Assignment operations are
x = 2; // idempotent
x = 2;
x = 2; // Calling it multiple times
x = 2; // always produces the same result
```

`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**

`strtok()` vs `strsep()`

- Alternatively, you can use `strsep()`
- 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 pages) 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 5
- If the keyword is unknown,
  - It's probably the name of an executable
  - So run it!
    - `fork()` and friends

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!
  - Refer to lab 5 on how to dynamically load plugins
- If the keyword is unknown,
  - It's probably the name of an executable
  - So run it!
    - `fork()` and friends

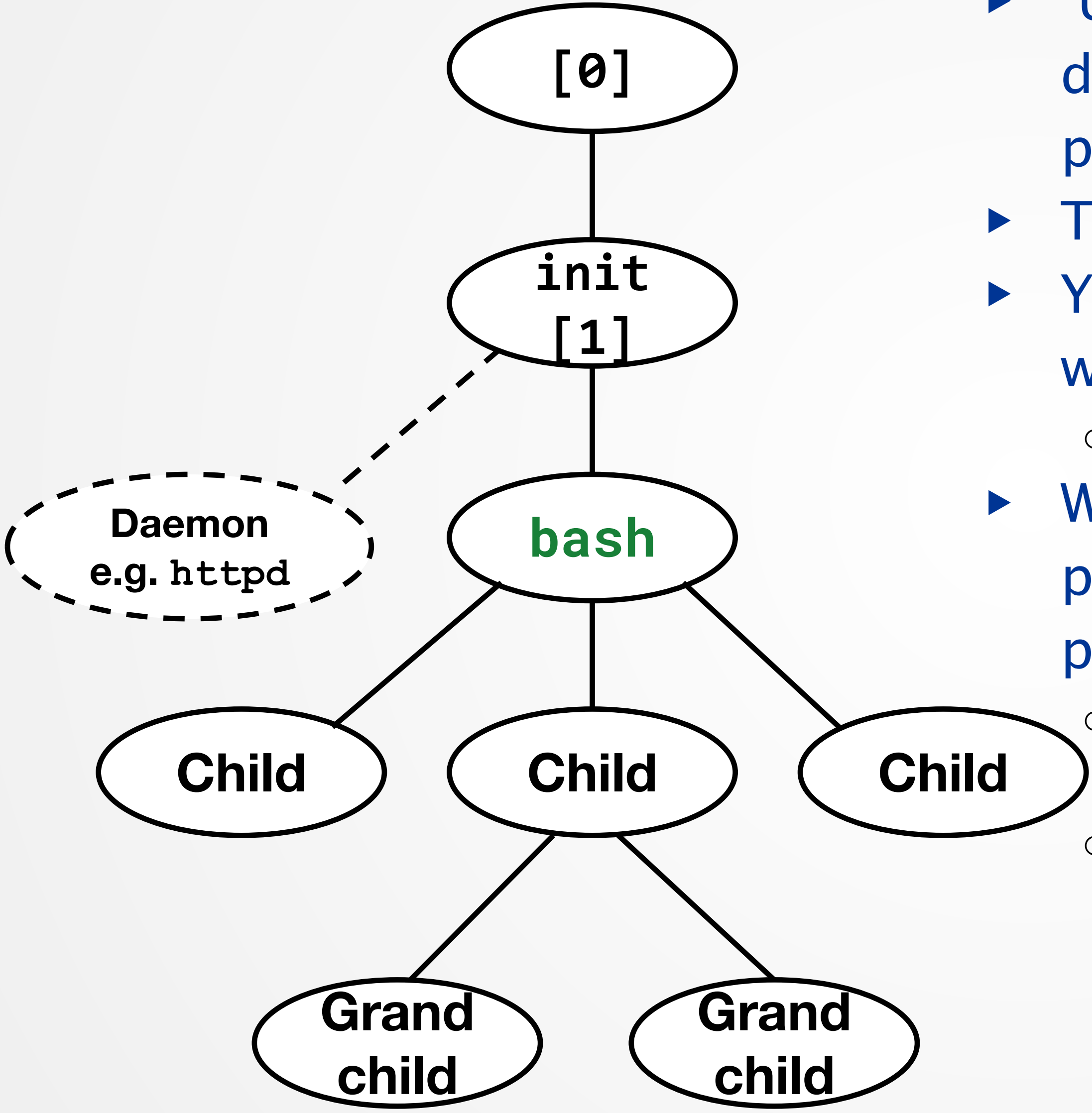
Building the shell: Executing command

- 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 5)
     -

Building the shell: Executing command

- Once we've tokenized the input, we can use standard C-string functions to *compare*
- 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…must be an executable name!

UNIX Process Hierarchy



- UNIX/Linux has an interesting design: every application is a child process.
- The root is the init task.
- Your shell spawns child processes when you ask to run a command.
  - Using `fork()` and `exec()`!
- When your own application spawns a process, it spawns its own child process using `fork()`
  - If your app exits before the child, the child becomes an orphan 😢
  - Orphan processes are *adopted* by the root

Building the shell: Executing command

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

Implementation Challenges & Hints

1. 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
2. When multiprogramming with `fork()`s
   - Think about the order in which processes need to run
   - Does a process need to wait for another?
3. 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

➢ Since this project requires access to many standard library functions, we highly recommend developing on Thoth or another Linux machine
   - And plan for outages!
     i. Back-up frequently (to your local machine)

Works Referred

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