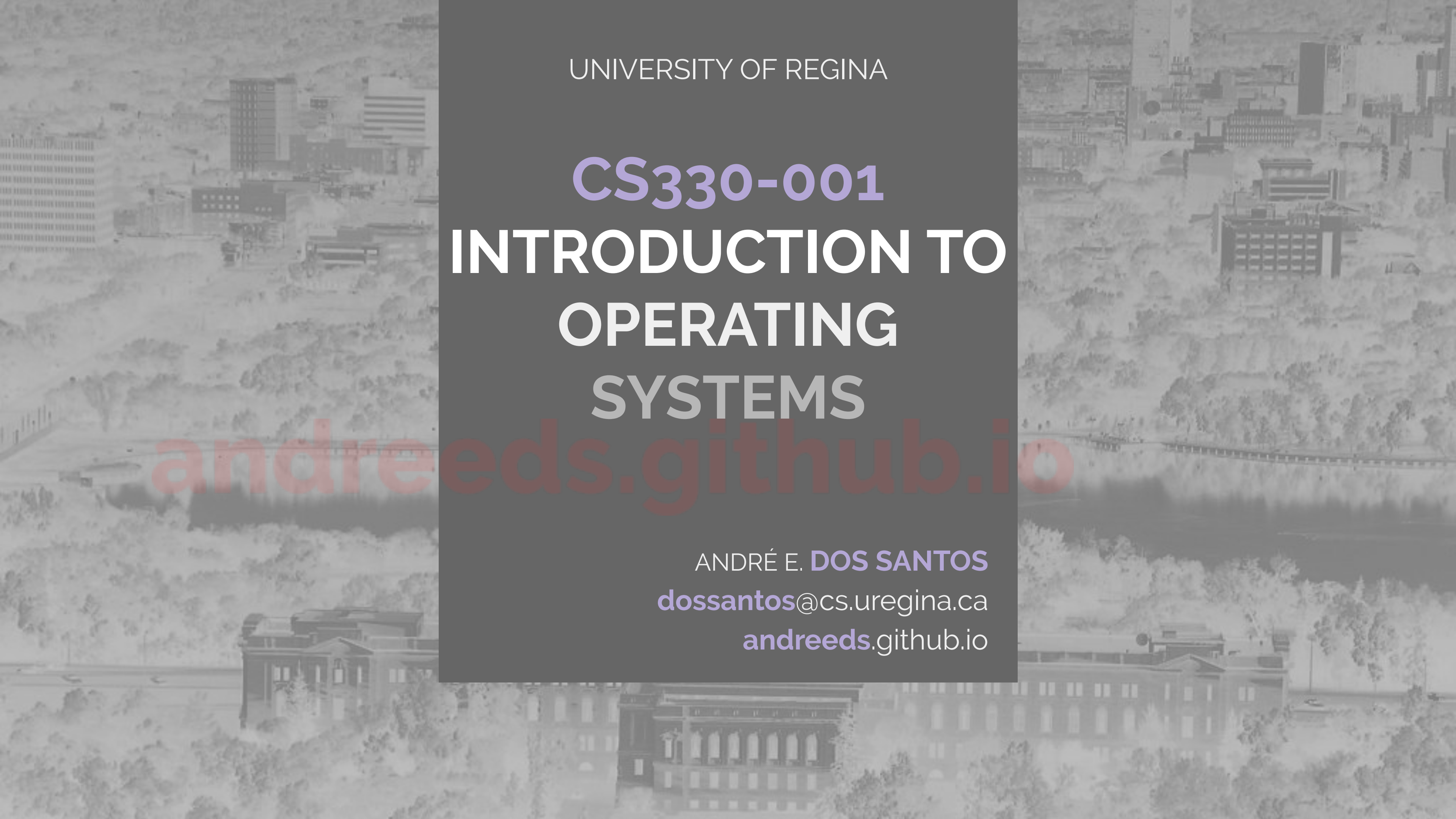
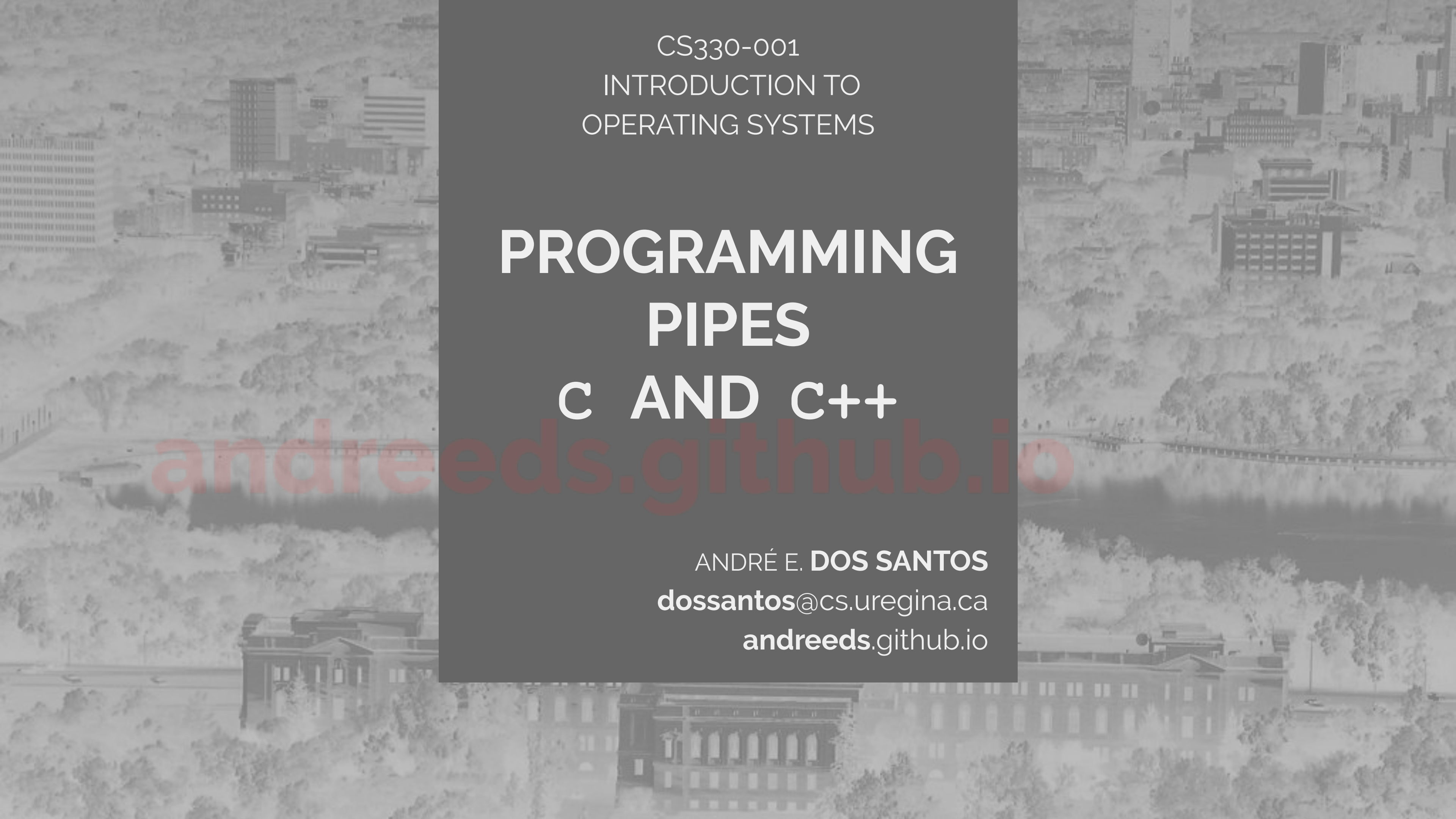


UNIVERSITY OF REGINA

CS330-001 INTRODUCTION TO OPERATING SYSTEMS

ANDRÉ E. DOS SANTOS
dossantos@cs.uregina.ca
andreeds.github.io



CS330-001
INTRODUCTION TO
OPERATING SYSTEMS

PROGRAMMING PIPES C AND C++

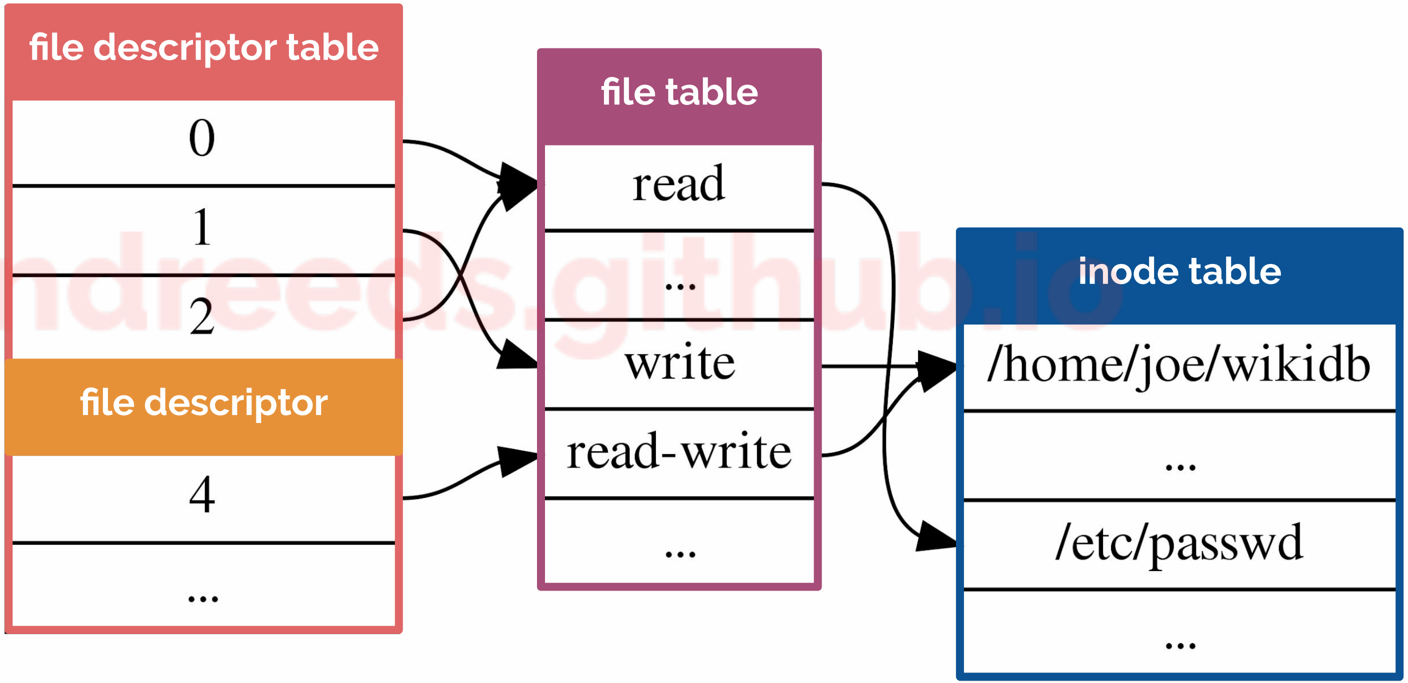
ANDRÉ E. **DOS SANTOS**
dossantos@cs.uregina.ca
andreeds.github.io

BASIC CONCEPTS IN UNIX FILE I/O

File descriptors are a fundamental component in accessing physical files from a process

- A **file descriptor** (i.e., a process' idea of a file) is a **positive integer** representing the **offset** (i.e., **index**) into a **process' file descriptor table**
- The **file descriptor table** provides a pointer to an element in the system file table (a kernel data structure)
- The **system file table** has one element for each currently **active open**, where each element provides a pointer to an element in the **in-memory inode table**
  - shared by all processes in the system
  - several system file table entries may point to the same element in the **in-memory inode table**
- The **in-memory inode table** contains one entry for each **active file** and provides a **map to the physical file**
  - a kernel data structure

BASIC CONCEPTS IN UNIX FILE I/O



file descriptor table

0

1

2

file descriptor

4

...

file table

read

...

write

read-write

...

inode table

/home/joe/wikidb

...

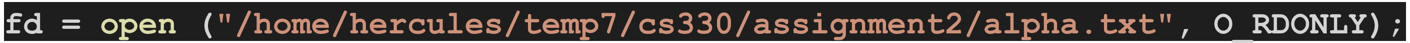
/etc/passwd

...

BASIC CONCEPTS IN UNIX FILE I/O

Example

- Let's assume some process executes the following statement



```
fd = open ("/home/hercules/temp7/cs330/assignment2/alpha.txt", O_RDONLY);
```

- Assume the process has no other open files
- Upon execution, the open function creates an entry in the file descriptor table and returns the value 3 indicating the file descriptor's index in the file descriptor table
- The entry in the file descriptor table points to the newly created entry in the system file table
- If the file is not currently in use by another process, the entry in the system file table points to a newly created entry in the in-memory inode table
    - Otherwise, the entry in the system file table points to a previously created entry in the in-memory inode table

BASIC CONCEPTS IN UNIX FILE I/O

- The **file descriptor table** actually also contains elements corresponding to each of
  - **standard input** - usually the keyboard
  - **standard output** - usually the screen
  - and **standard error** - usually the screen
- Although standard input, standard output, and standard error are not technically files, it just so happens that **UNIX treats all devices like files**
  - the name associated with every device will look like a pathname

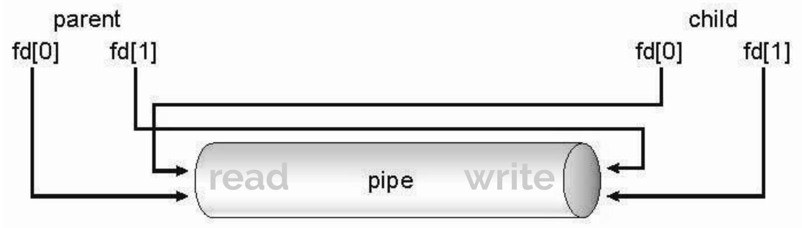
`int`	**name**	`<unistd.h>`	`<stdio.h>`
`0`	standard input	`STDIN_FILENO`	`stdin`
`1`	standard output	`STDOUT_FILENO`	`stdout`
`2`	standard error	`STDERR_FILENO`	`stderr`

PIPES

- A **pipe** provides a one-way flow of data from the write end to the read end (the read end and the write end actually correspond to a special file represented by two file descriptors stored as ints).

```
int fd [2];       /* or any other valid variable name */
```

  - The **pipe** system call
    - finds the first **two open positions** in a process' **file descriptor table**
    - and allocates them for the **read and write end of the pipe**, respectively
  - **Data is read from** `fd [0]` on a FIFO basis
  - **Data is written from** `fd [1]`



PIPE SYSTEM CALLS

The pipe system call creates a unidirectional communication buffer that can be accessed through the file descriptors.

```
#include <unistd.h>
int pipe (int *file_descriptor);
```

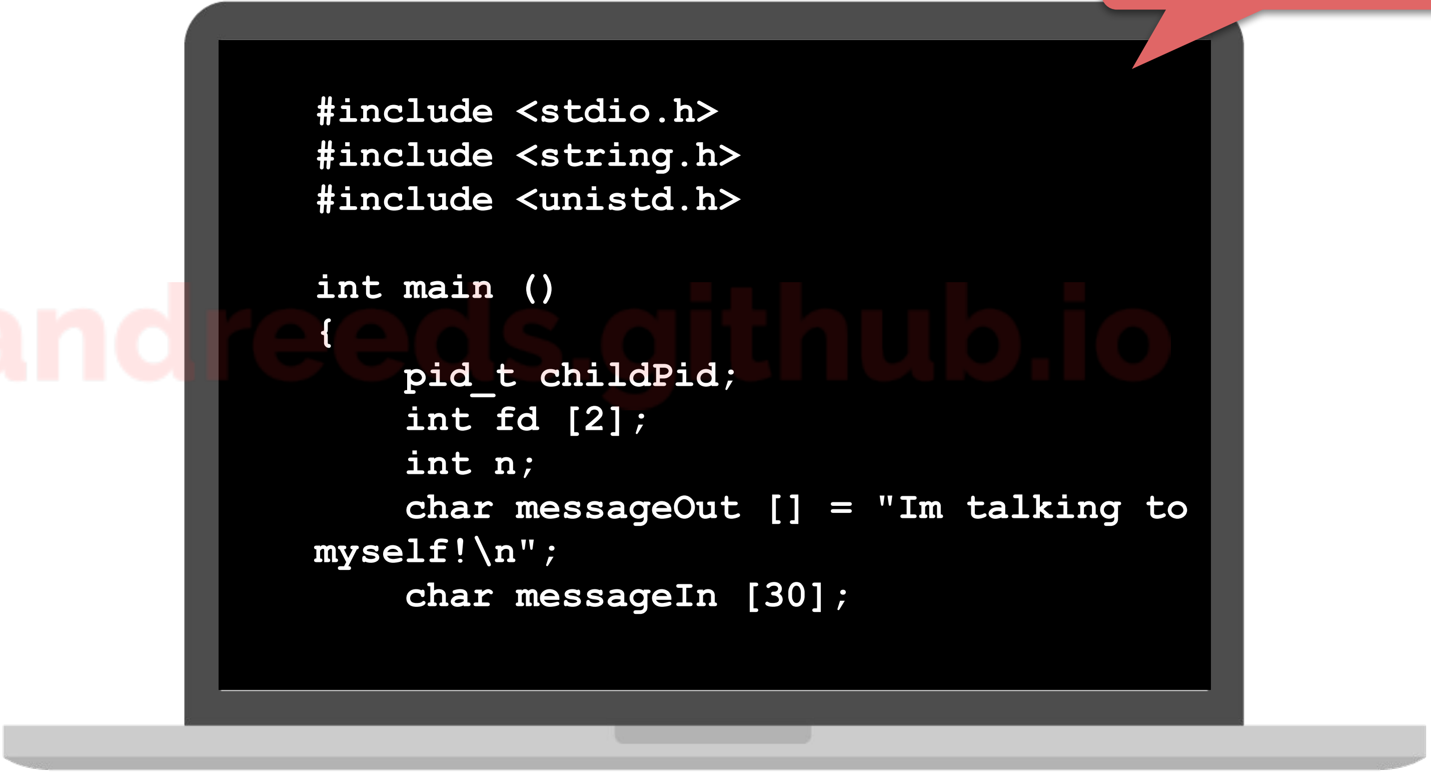
- If successful, pipe returns **0**
- If unsuccessful, pipe returns **-1** and sets errno

- A pipe has **no** external or permanent name
  - it must be accessed through the **file descriptors**
  - restricts its use to the process that created it and any child processes that inherit the descriptors following a fork system call

PIPE SYSTEM CALLS

oneProcessWriteReadPipe.c

URCourses



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

int main ()
{
    pid_t childPid;
    int fd [2];
    int n;
    char messageOut [] = "Im talking to
myself!\n";
    char messageIn [30];
```

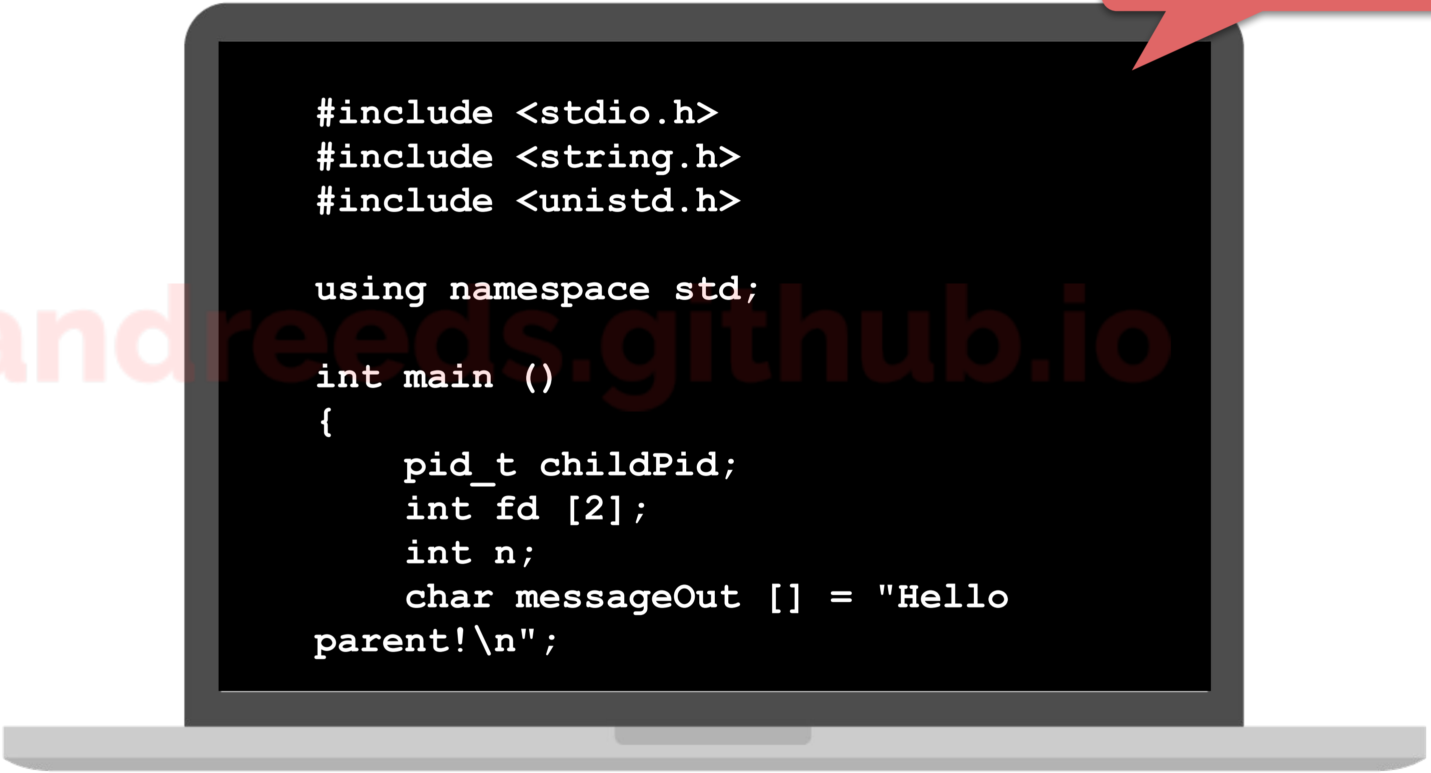
PIPES

- Pipes are typically used to communicate between two different processes
- When messages are to be sent **from a parent process to a child process (or vice-versa)**, the pipes are set up according to the following sequence of events:
  1. **parent `creates` a pipe**
  2. **parent `forks` a child**
  3. ***producer process* `closes` the read end** of the pipe
  4. ***consumer process* `closes` the write end** of the pipe

PIPE SYSTEM CALLS

childWritePipe.c

URCourses



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

using namespace std;


int main ()
{
    pid_t childPid;
    int fd [2];
    int n;
    char messageOut [] = "Hello
parent!\n";
```

PIPES

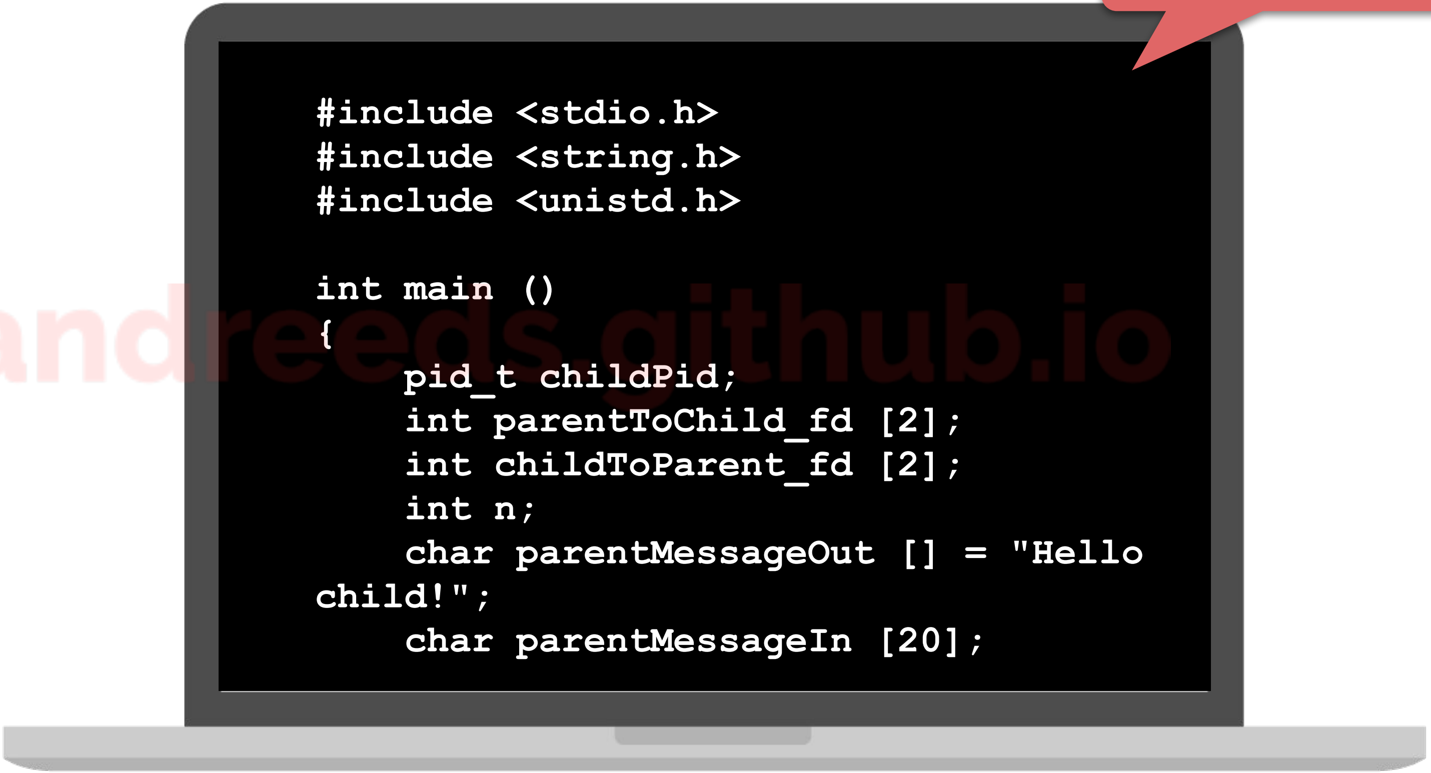
- Since the data flow in a pipe is **unidirectional**, **two pipes** are required for data flow that is bidirectional
- When messages will be exchanged by the parent and child, the pipes are set up according to the following sequence of events:

1. parent `creates` two pipes:
   i. one for parent-to-child messages and
   ii. one for child-to-parent messages
2. parent `forks` a child
3. parent `closes` the read end of the parent-to-child pipe
4. parent `closes` the write end of the child-to-parent pipe
5. child `closes` the read end of the child-to-parent pipe
6. child `closes` the write end of the parent-to-child pipe

PIPE SYSTEM CALLS

twoWayPipe.c

URCourses



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

int main ()
{
    pid_t childPid;
    int parentToChild_fd [2];
    int childToParent_fd [2];
    int n;
    char parentMessageOut [] = "Hello
child!";
    char parentMessageIn [20];
```

CHARACTERISTICS OF I/O USING PIPES

- When a process tries to read from a pipe the **`read`** system
  - call returns immediately if the pipe is not empty
  - call blocks and remains blocked until something is written to the pipe.
- When a process tries to write to a full pipe, the **`write`** system
  - call blocks until sufficient data has been read from the pipe to allow the write to complete

- When a process tries to read from a pipe, if no process has the pipe open for writing, the read system call returns **`0`**
  - i.e., end-of-file
- When a process tries to write to a pipe, if no process has the pipe open for reading, the write system call will return **`-1`**
  - i.e., it fails

CHARACTERISTICS OF I/O USING PIPES

- **A pipe has a limited capacity**
  - If the pipe is full, then as previously mentioned, a write system call will block
  - Different implementations have different limits for the pipe capacity, so applications should not rely on a particular capacity
  - **Design tip**: A reading process should be designed to **read data** from the pipe as **soon** as it is available in order **to prevent a writing process from blocking**

UNIX FILTERS

- A **filter** reads from standard input, performs some **transformation**, and writes to **standard outpu**t
  - e.g., **`head`**, **`tail`**, **`cat`**, **`more`**, **`sort`**, and **`grep`**
- A fundamental characteristic of a filter is that it **should not require any interaction with the user**

UNIX FILTERS

Example 1

- The **`cat`** command with no command-line arguments reads from standard input and writes to standard output. But using output redirection, cat can write to a file

```
% cat > junk.txt
```

- Whatever is typed at the keyboard will be written to the file
- The redirection occurs because the shell changes the standard output entry of the file descriptor table to the system file table entry associated with **`junk.txt`**

UNIX FILTERS

- Now recall that a **file descriptor** is an **index** into the **file descriptor table**
- Using an action known as redirection, a program can **modify the file descriptor table entry** so that it **points to a different entry** in the **system file table**

PIPE SYSTEM CALLS

The `dup2` system call is used to redirect I/O by copying a file descriptor from one entry in the file descriptor table to another.

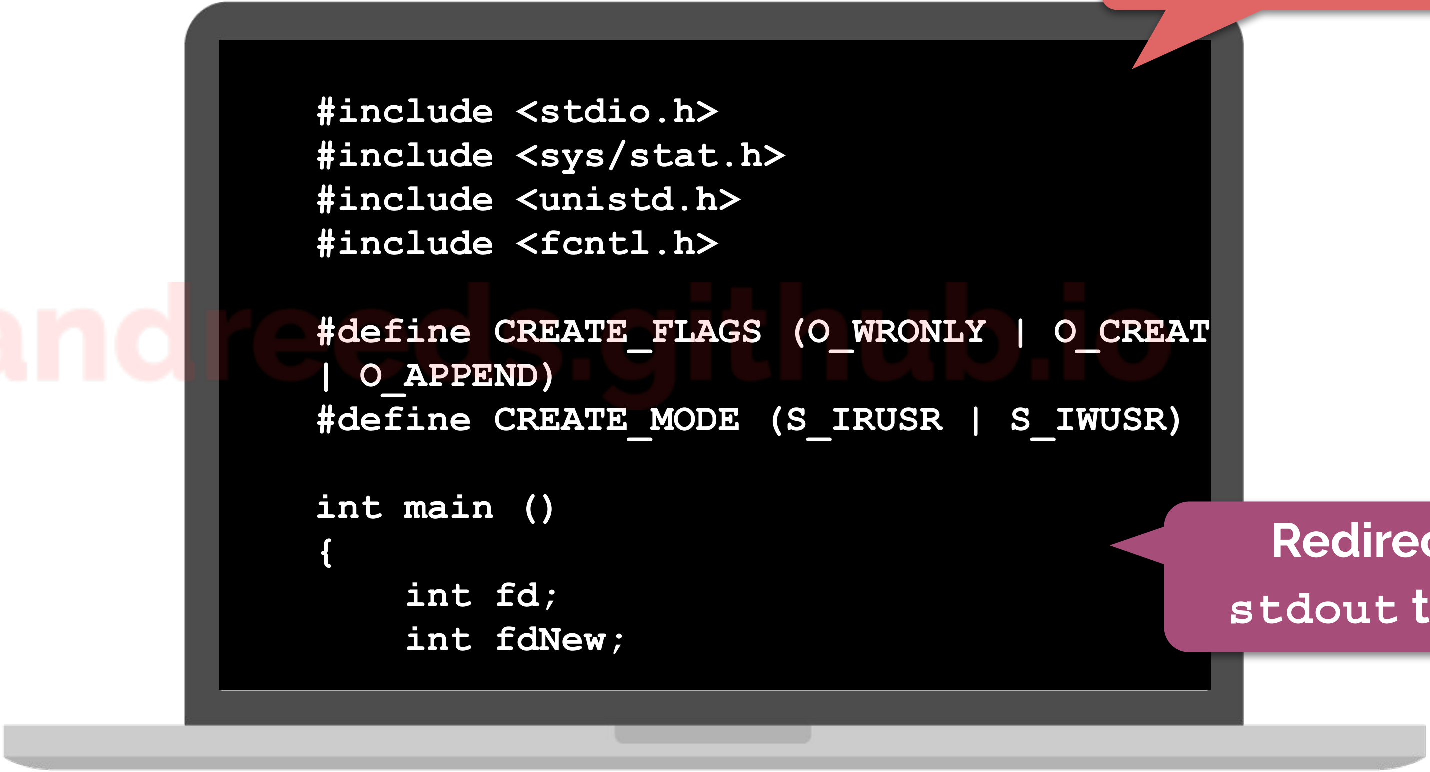
```
#include <unistd.h>
int dup2 (int fd1, int fd2);
```

- The `dup2` system call closes the entry for `fd2` in the file descriptor table if it was open and then copies the pointer of the entry for `fd1` into the entry for `fd2`
- Although `dup2` closes `fd2`, some think that a responsible programmer would close `fd2` before calling `dup2`
- If successful, `dup2` returns the index for `fd2` in the **file descriptor table**
- If unsuccessful, `dup2` returns **–1** and sets `errno`

PIPE SYSTEM CALLS

redirectStdoutToFile.c

URCourses



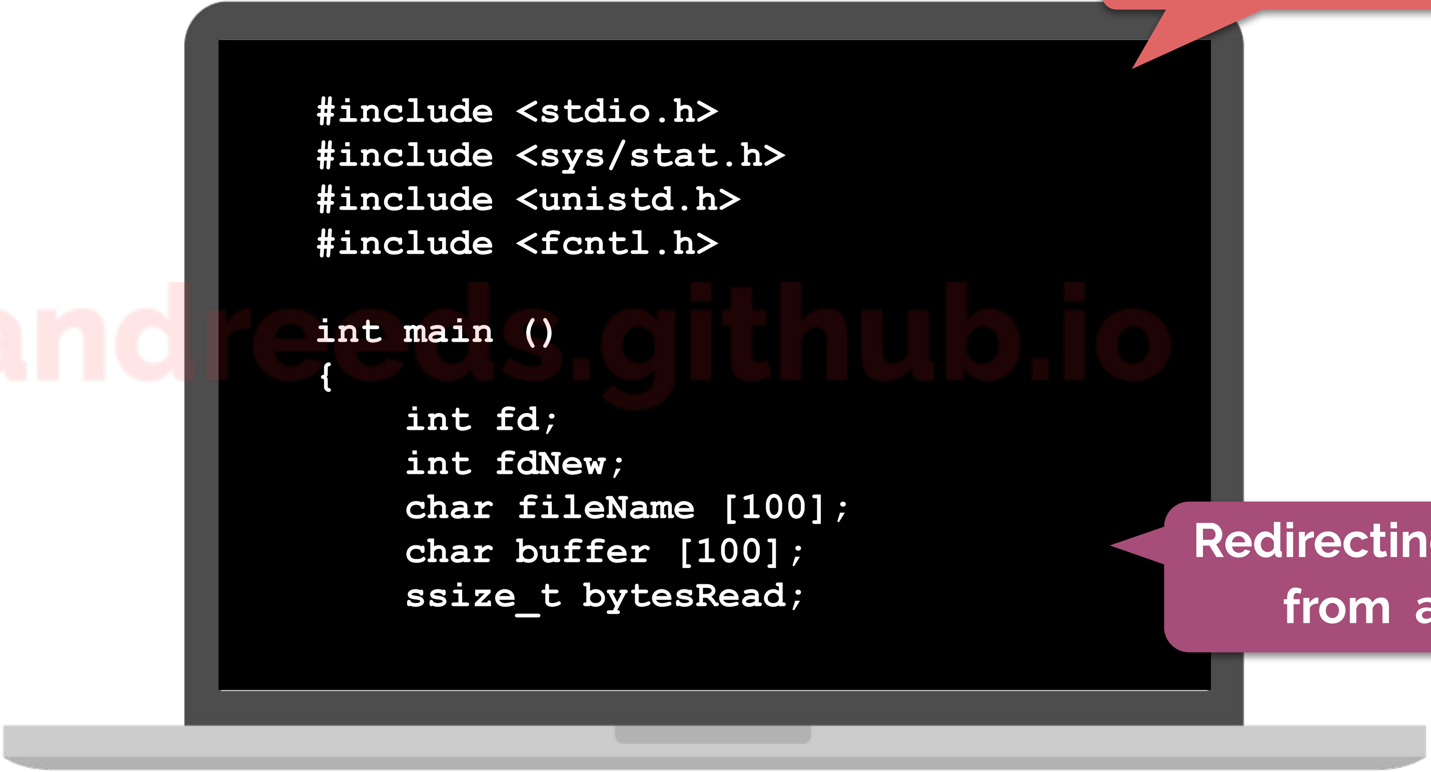
```
#include <stdio.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>

#define CREATE_FLAGS (O_WRONLY | O_CREAT
| O_APPEND)
#define CREATE_MODE (S_IRUSR | S_IWUSR)

int main ()
{
    int fd;
    int fdNew;
```

Redirecting `stdout` to a file

PIPE SYSTEM CALLS

`redirectStdinFromFile.c`

URCourses



```
#include <stdio.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>

int main ()
{
    int fd;
    int fdNew;
    char fileName [100];
    char buffer [100];
    ssize_t bytesRead;
```

Redirecting `stdin` from a file

UNIX FILTERS

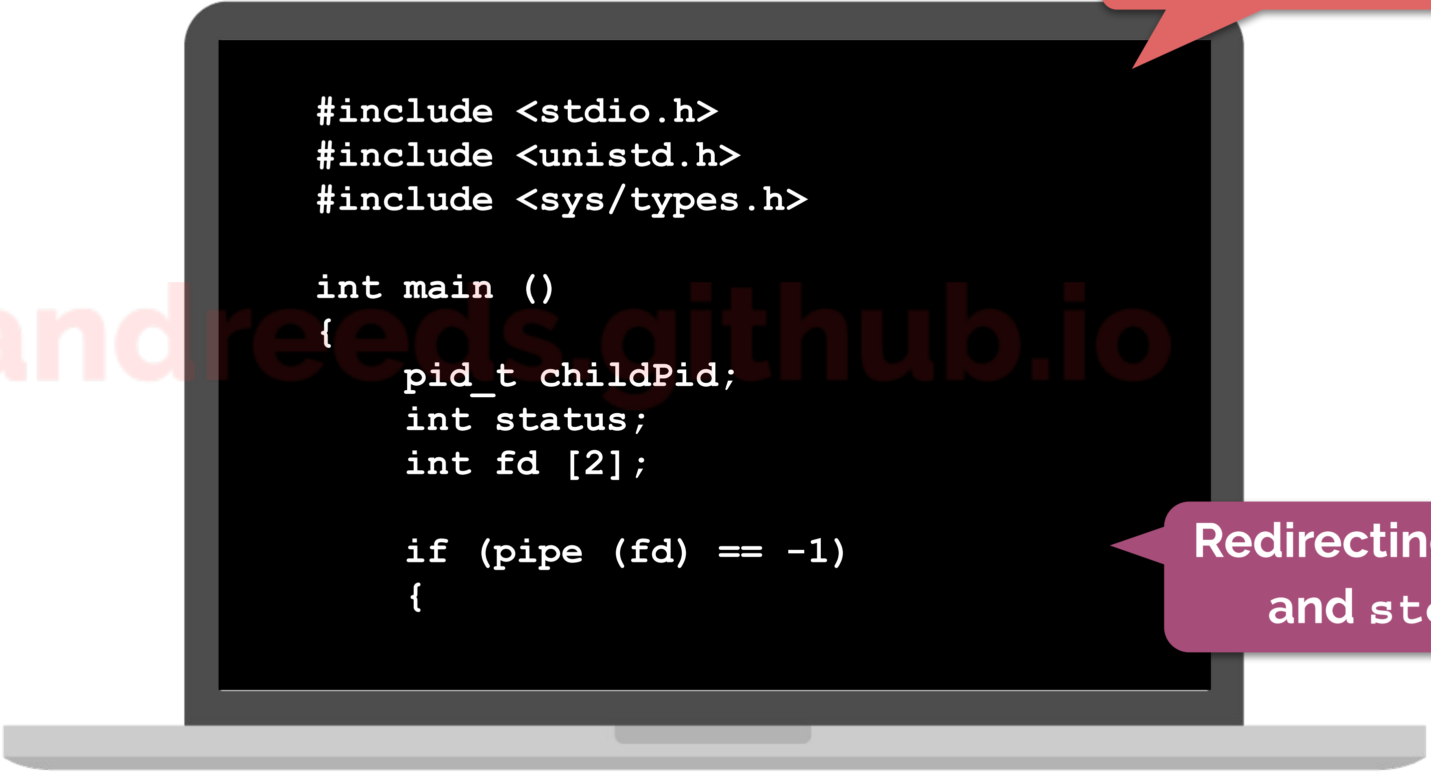
Pipelines are groups of programs that use redirection with pipes to connect processes together

```
% ls > junk.txt
% wc -l < junk.txt
```

PIPE SYSTEM CALLS

`redirectStdioUsingPipe.c`

URCourses



```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main ()
{
    pid_t childPid;
    int status;
    int fd [2];

    if (pipe (fd) == -1)
    {
```

Redirecting `stdin` and `stdout`