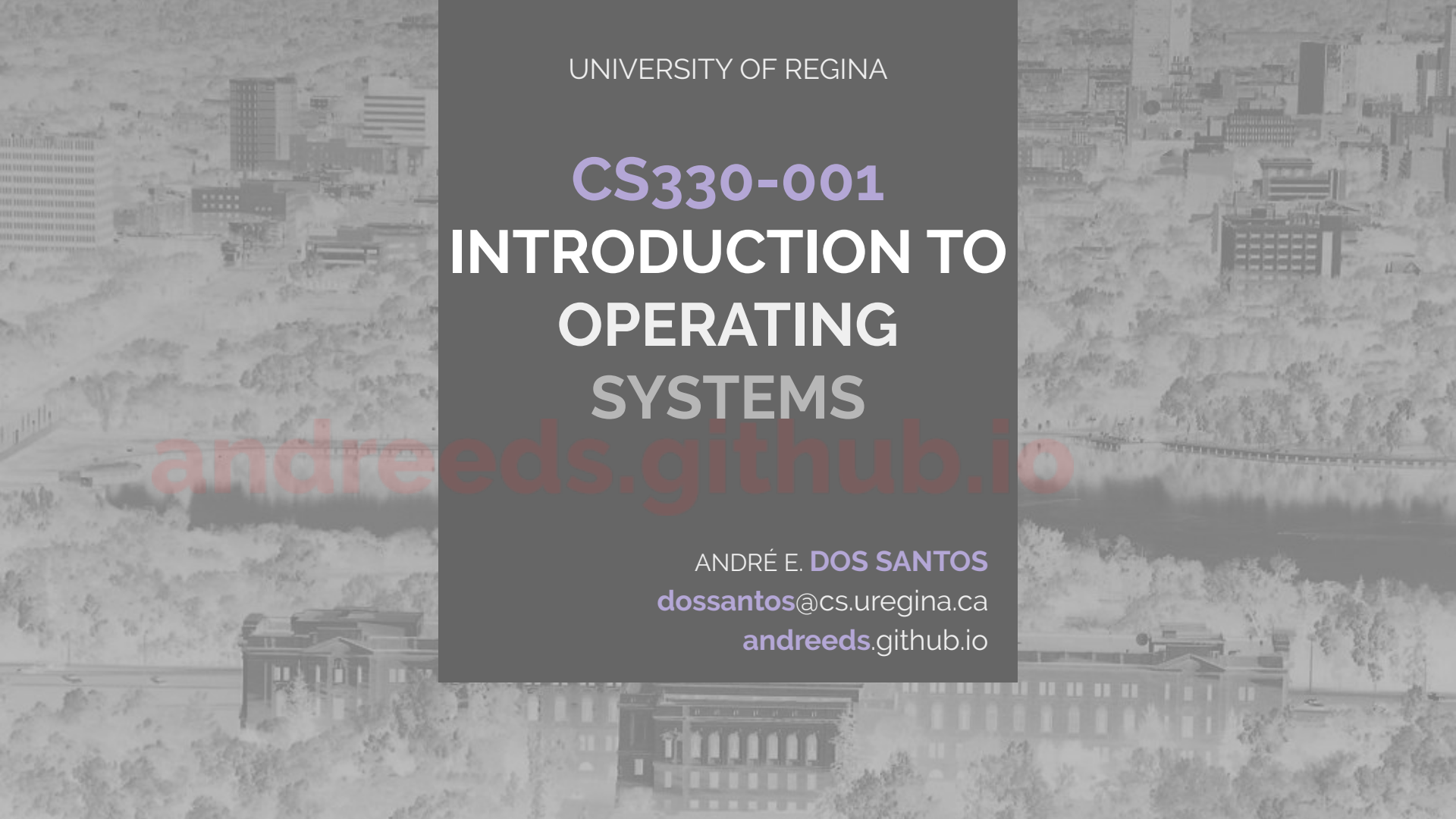

UNIVERSITY OF REGINA

CS330-001
INTRODUCTION TO OPERATING SYSTEMS

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

CS330-001
INTRODUCTION TO
OPERATING SYSTEMS

PROGRAMMING PROCESSES

C++

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

PROCESSES

PROCESSES

- **process ID** - A unique identifier given to a process by the OS
- All relevant information that defines the state for a given process can be stored in a data structure or structures by the OS
- When a context when a **context switch occurs**
  - the state information for the process to be **suspended** is saved by the OS
  - the state information for the process to be **executed** is restored and
  - the restored process begins execution

CREATING A PROCESS

- **child process** - A new process that is created by a currently executing process (the parent process)
- **parent process** - The currently executing process that has created one or more new child processes
- A new process is created by an existing process by calling the **`fork`** function

CREATING A PROCESS

Example

```
//  Parent process
cout << "Before fork" << endl;
pid = fork();
cout << "After fork" << endl;
```


- The value returned to `pid` (data type `pid_t`) is the ID of the new process
- Data type `pid_t` is an `int` type that is used to represent the process ID
- The process child is created as a copy of the parent process

CREATING A PROCESS

- Because the child process is a copy of the parent to process the program counters of both processes have to same value
- The next statement to execute in both processes would be the insertion to `cout` following the call to `fork`

Parent Process

```
...
cout << "Before fork" << endl;
pid = fork()
cout << "After fork" << endl;
...
```

Child Process

```
...
cout << "Before fork" << endl;
pid = fork()
cout << "After fork" << endl;
...
```

- Keep in mind that the insertion insertion to `cout` in a particular process does not execute until the OS gives control to that process

CREATING A PROCESS

- A parent process can have more than one child process, and a child process can have its own child processes
- If the `fork` function is **not successful**
  1. **returns to the parents process**
     - with value `-1`
  2. **the child process is not created**

CREATING A PROCESS

- If the `fork` function creates a new process and it **successful**
  1. **returns to the parents process**
     - **the process ID** saved of the newly created **child process**
       - saved in `pid` for the parent process
  2. **returns to the child process** it created
     - **value** `0`
       - saved in `pid` for the child process

PARENT AND CHILD PROCESSES WITH DIFFERENT BEHAVIORS

- In the following fragment the `fork` function creates a new process returning to process ID of the child process to the parent process and 0 the child process
- The value of `pid` is used in the conditional statements to cause different blocks of code to execute in each process

PARENT AND CHILD PROCESSES WITH DIFFERENT BEHAVIORS

```
#include <iostream>
#include <unistd.h>
...
pid_t pid;
...
pid = fork();
...
if (pid < 0){
  cout << "Error creating new process" << endl;}
else if (pid == 0){
  cout << "Child ID " << getpid() << endl;}
else {
  cout << "Parent ID " << getpid() << " Child ID " << pid << endl;}
```

SOME PROCESSES FUNCTIONS FROM unistd.h AND wait.h

Function	Example	Parameters	Result type
fork	`pid = fork();`	*None*	`pid_t`
getpid	`pid = getpid();`	*None*	`pid_t`
wait	`pid = wait(&status_ptr);`	`int* status_pts`	`pid_t`

execl	`execl("prog.exe", "prog.exe", NULL);`	`const char *path`
		`const char *file`
		…
		`NULL`
			`int`

WAITING FOR A PROCESS

- The process can exit independently of its parent or child process
- When a child process exits before its parents process
  - the parent process can retrieve the **exit status** of the child process
    - to determine whether the child process exited successfully or not
- The parent process can retrieve the exit status of a child process by calling to function `wait`

WAITING FOR A PROCESS

```
#include <iostream>
#include <unistd.h>
...
pid_t pid;
int status;
...
pid = fork();
...
// Pause until a child process exits
pid = wait(&status);
if (pid > 0){
  cout << "pid " << pid << " status " << WEXITSTATUS(status) << endl;}
...
```

WAITING FOR A PROCESS

- The `wait` function returns
  - **the process ID of the child process that exited the function result**
  - **or `0` if there are no child processes remaining**
- The `wait` function also stores the exit status of the process that exited
  - including the exit code, in the function argument `status` (type `int`)
- The macro `WEXITSTATUS(status)` call be used to extract the exit status can be used to extract the exit cold from the variable `status`
- In the fragmenting, the insertion to `cout` executes after the return from the `wait` function

EXECUTING ANOTHER PROGRAM FROM A PROCESS

- Most often be want to create a new process that will perform a different function
- To do this, we need to create a new process and then to replace the instructions associated with the new process (using fork) with instructions perform the desired operation

EXECUTING ANOTHER PROGRAM FROM A CHILD PROCESS

```
#include <iostream>
#include <unistd.h>
...
pid_t pid;
...
pid = fork();
...
if (pid == 0)
  {
  execl("newprogram.exe", "newprogram.exe", NULL);
  }
cout << "In Parent process after if statement" << endl;
...
```

EXECUTING ANOTHER PROGRAM FROM A PROCESS

```
#include <iostream>
#include <unistd.h>
...
pid_t pid;
...
pid = fork();
...
execl("newprogram.exe", "newprogram.exe", "Arg1", "Arg2","ArgN", NULL);
cout << "Error Reading/Executing the File newprog.exe" << endl;
...
```

EXECUTING ANOTHER PROGRAM FROM A PROCESS

- The instructions in the current process are replaced by the instructions from the executable file newprog.exe which is assumed to be in the current working directory, with the arguments Arg1, Arg2, and ArgN passed to the new instructions as if they were typed in the command line:
  - `newprog.exe Arg1 Arg2 ArgN`