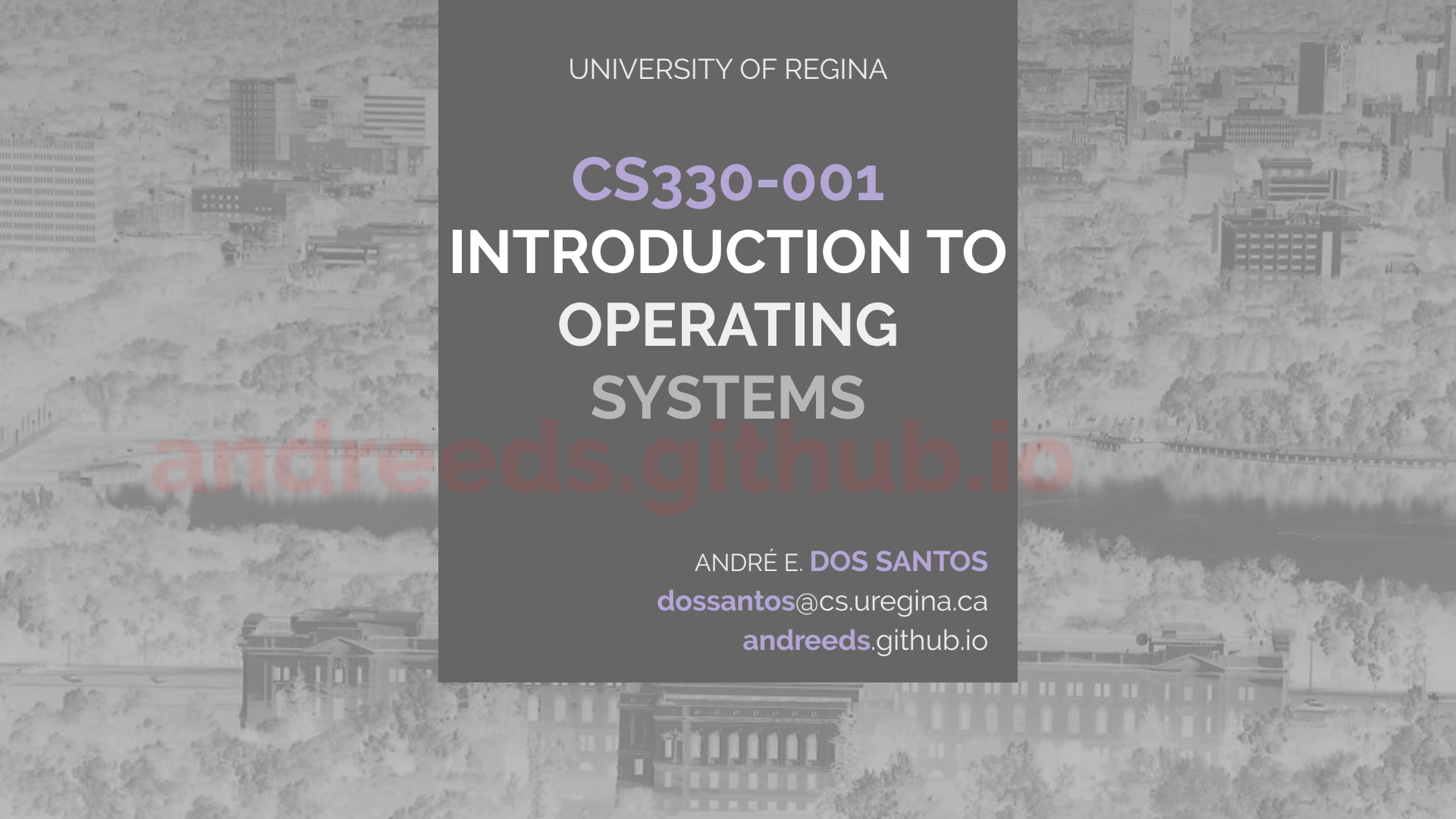


UNIVERSITY OF REGINA

# 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

# UNIX/LINUX PROCESSES

# THE `fork` FUNCTION

- A process can create a new process by calling the `fork` function

```
#include <unistd.h>
pid_t fork (void);
```

  - The calling process becomes the *parent*
  - The new process is called the *child*
  - The parent and child have different process IDs
  - The `pid_t` type is an unsigned integer
  - In the parent, the call to `fork` returns the process ID of the child process if the creation of the child process was successful. Otherwise, it returns **–1**
  - In the child, the call to `fork` always returns **0**.
  - The parent and child have their own separate PCBs, so do not share any data through like-named variables
  - Both the parent and child continue at the instruction following the call to `fork`

# THE `getpid` AND `getppid` FUNCTION

- The `getpid` and `getppid` functions return the process ID and parent process ID, respectively

```
#include <unistd.h>
pid_t getpid (void);
pid_t getppid (void);
```

  - Neither `getpid` nor `getppid` can return an error.

## simpleFork.c

URCourses

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

int main ()
{
    int x = 0;

fork ();
x = 1;
printf ("I am process %d, ", getpid ());
printf ("my parent is %d, ", getppid ());
printf ("and my x is %d.\n", x);

return 0;
}
```

## betterFork.c

URCourses

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

int main ()
{
    pid_t childPid = 0;
    int x = 0;

    childPid = fork ();
    if (childPid == -1)
    {
        fprintf (stderr, "Process %d failed to
fork!\n", getpid ());
        return 1;
    }
```

# CHAIN AND FAN

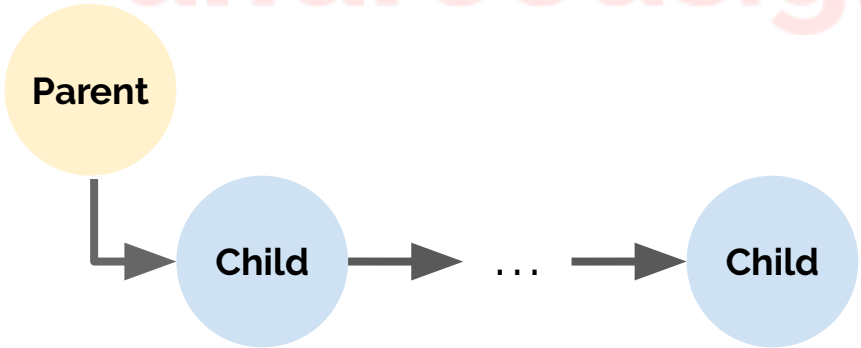
## Chain

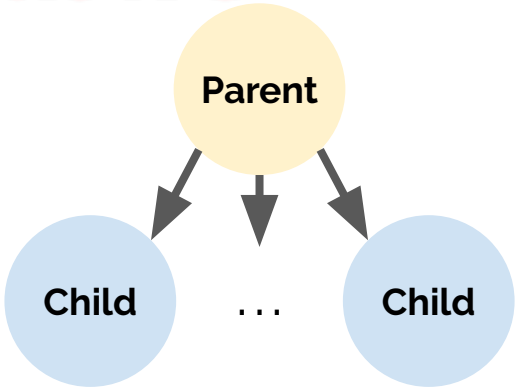
```
pid_t childpid = 0;
for (i=1;i<n;i++)
   if (childpid = fork())
       break;
```

## Fan

```
pid_t childpid = 0;
for (i=1;i<n;i++)
  if (childpid = fork() <= 0)
      Break;
```





## processChain.c

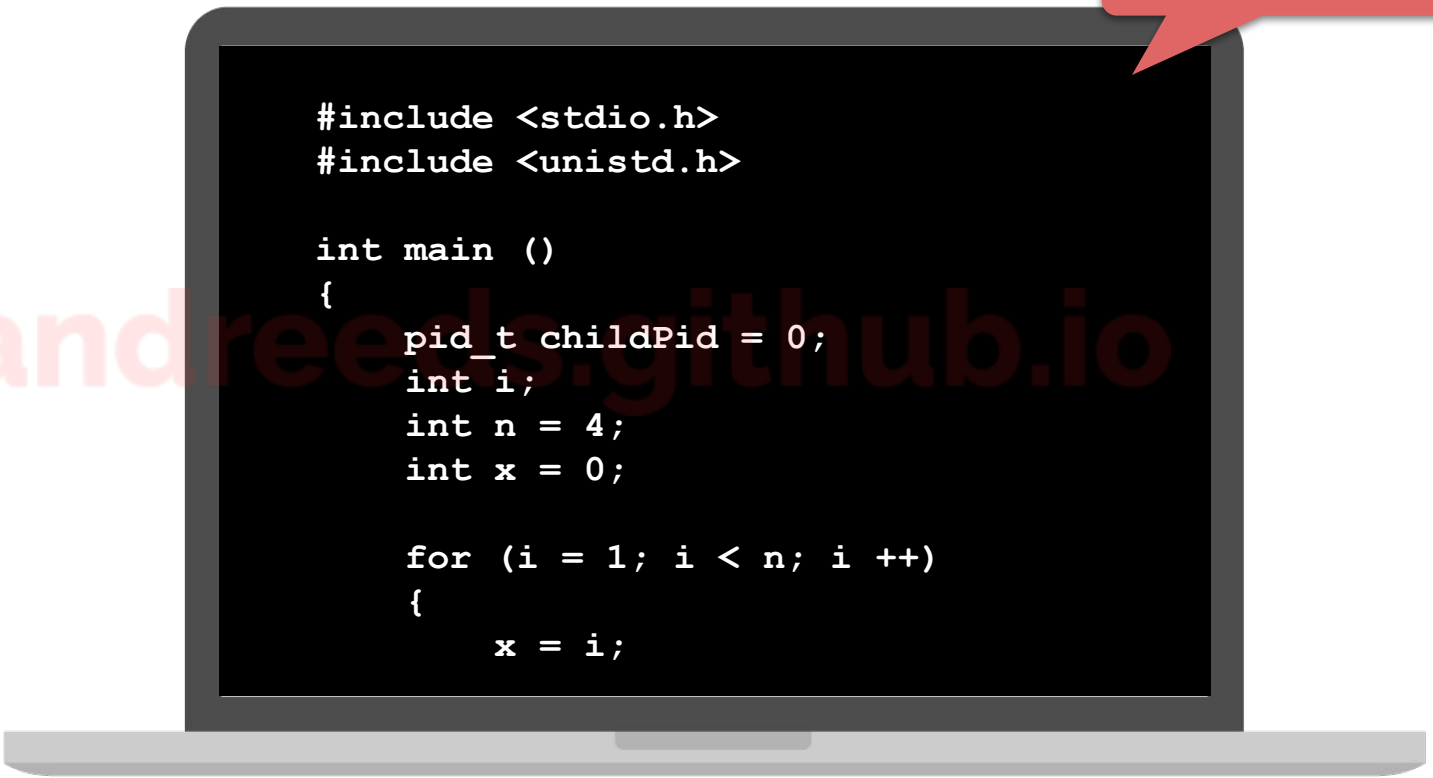
URCourses

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

int main ()
{
    pid_t childPid = 0;
    int i;
    int n = 4;
    int x = 0;

    for (i = 1; i < n; i ++)
    {
        if (childPid = fork ())
```

# UNIX/LINUX PROCESSES

## processFan.c

URCourses



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

int main ()
{
    pid_t childPid = 0;
    int i;
    int n = 4;
    int x = 0;

    for (i = 1; i < n; i ++)
    {
        x = i;
```


YOU ARE ABSOLUTELY FORBIDDEN TO EXPERIMENT WITH
PROGRAMS USING fork() ON HERCULES!!

# WHY?

- When using and experimenting with `fork`, you may set up a process that creates some number of child processes, and then each of these processes creates some number of child processes, and then each of these processes…
  - you get the picture
- Now let's assume that you have a bug in your program so that the creation of child processes is not going to stop
  - i.e., the termination condition is never satisfied
- Even if each process creates just two child processes, after one generation, you'll have two more processes, after two generations, four more, after three, eight more, …, after n, $2^n$
  - clearly, the number of processes created is growing exponentially

# WHY?

- These processes are created so fast that you will be unable to kill them fast enough to prevent Hercules from becoming overwhelmed
- Essentially, as soon as some number of processes is killed, the newly available resources will quickly be overwhelmed with new processes
  - This is called a `fork` BOMB!!

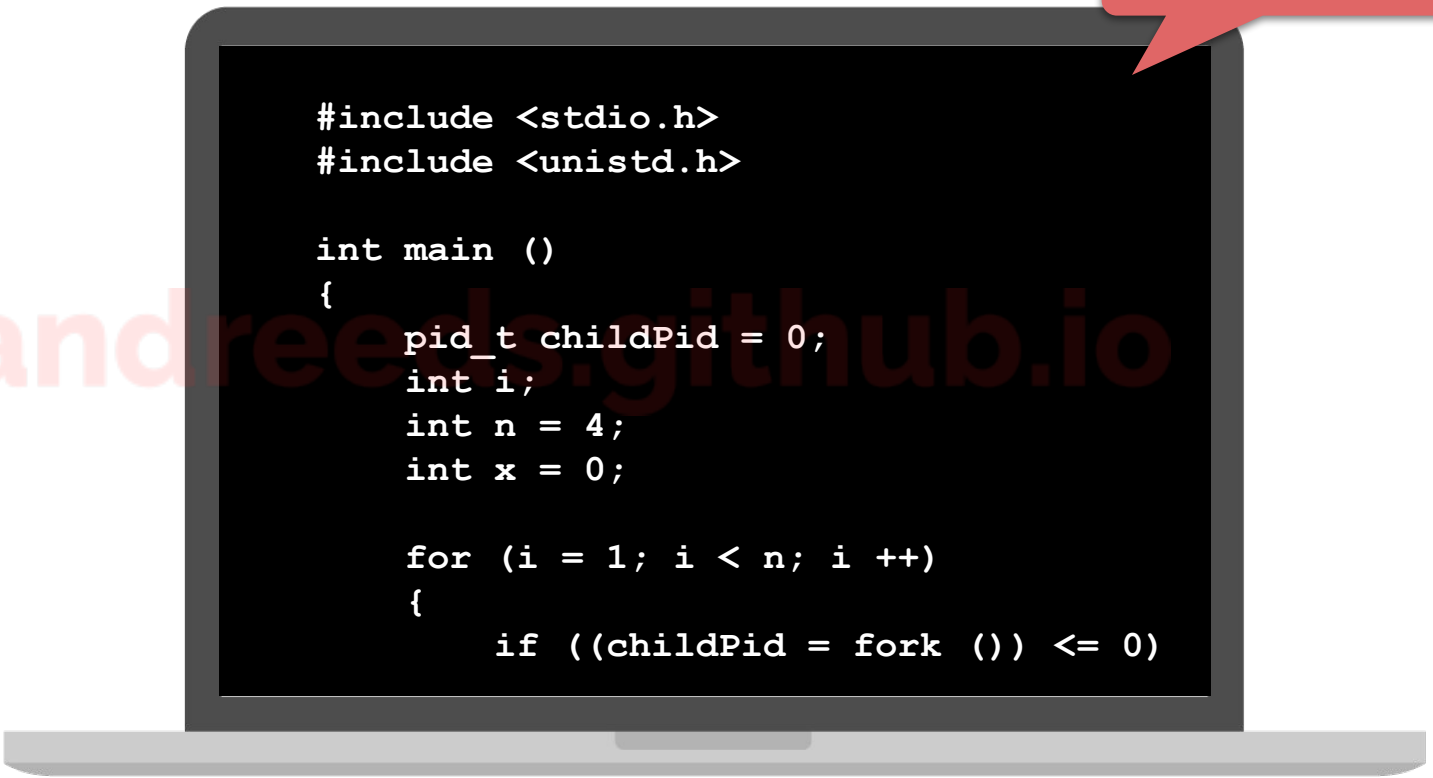
- **SO, WHEN USING AND EXPERIMENT WITH `fork`, DO IT ON THE LINUX MACHINES ONLY!**
  - Beware, it may happen to YOU
  - A repeat performance could cause your computing privileges to be restricted.

# THE `sleep` FUNCTION

- `#include <unistd.h>`
- `unsigned sleep (unsigned seconds);`
  - The `sleep` function returns **0** after the specified time has elapsed.

## processSleepyFan.c

URCourses



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

int main ()
{
    pid_t childPid = 0;
    int i;
    int n = 4;
    int x = 0;

    for (i = 1; i < n; i ++)
    {
        if ((childPid = fork ()) <= 0)
```

# THE `wait` FUNCTION

- The wait and waitpid functions cause the parent to block until the child has finished executing

```
#include <sys/wait.h>
pid_t wait (int *status);
pid_t waitpid (pid_t pid, int *status, int options);
```

  - The `wait` function causes the parent to block until it receives a signal that **any** child has finished executing

# THE `wait` FUNCTION

- The `waitpid` function causes the parent to block until it receives a signal that a **particular** child has finished executing.
- The `status` parameter is a pointer to a location for returning the status.
- The `pid` parameter determines how the parent will respond.
  - If pid is –1, the parent waits for any child (i.e., waitpid behaves like wait if options is also set to 0).
  - If pid is greater than 0, the parent waits for the child whose process ID is pid.
- The `options` parameter determines how the parent will handle the call to the `waitpid` function
  - i.e., blocking or non-blocking

# THE `wait` FUNCTION

- The `wait` and `waitpid` functions return the process ID of the child whose status is being reported
- If an error occurs, they return **`-1`** and set `errno`

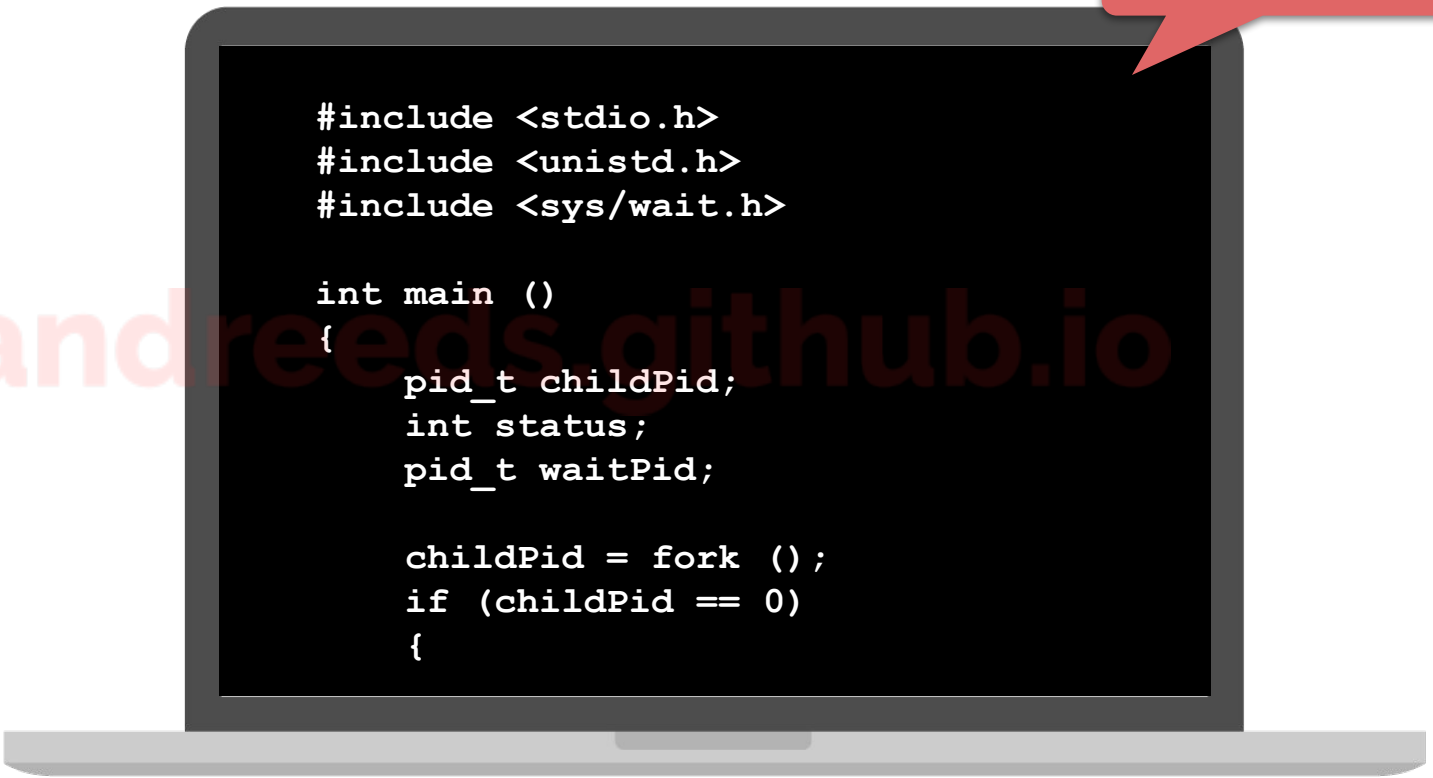
# THE `wait` FUNCTION

- The `wait` and `waitpid` functions return the process ID of the child whose status is being reported
- If an error occurs, they return **`-1`** and set `errno`

# UNIX/LINUX PROCESSES

## patientParentOfOne.c

URCourses



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

int main ()
{
    pid_t childPid;
    int status;
    pid_t waitPid;

    childPid = fork ();
    if (childPid == 0)
    {
```

# UNIX/LINUX PROCESSES

## patientParentOfMany.c

URCourses

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

int main ()
{
    pid_t childPid;
    int i;
    int n = 4;
    int status;
    pid_t waitPid;

    for (i = 1; i < n; i ++)
```

# ERROR HANDLING IN SYSTEM CALLS

- All system calls return **`-1`** if an error occurs
- There are a number of helpful facilities to
- diagnose the problem
  - the **`errno`** global variable

# ERROR HANDLING IN SYSTEM CALLS

- `errno`
  - A **global variable** called `errno` in every process
  - At process creation time, `errno` is initialized to **0**
  - When a system call error occurs, **errno** is set
  - See `/usr/include/sys/errno.h`
    - The `<errno.h>` header shall provide a declaration for `errno` and give positive values for the following symbolic constants
  - *Examples*
    - `EAGAIN = "Cannot fork process: System Process Limit Reached"`
    - `ENOMEM = "Cannot fork process: Out of memory"`

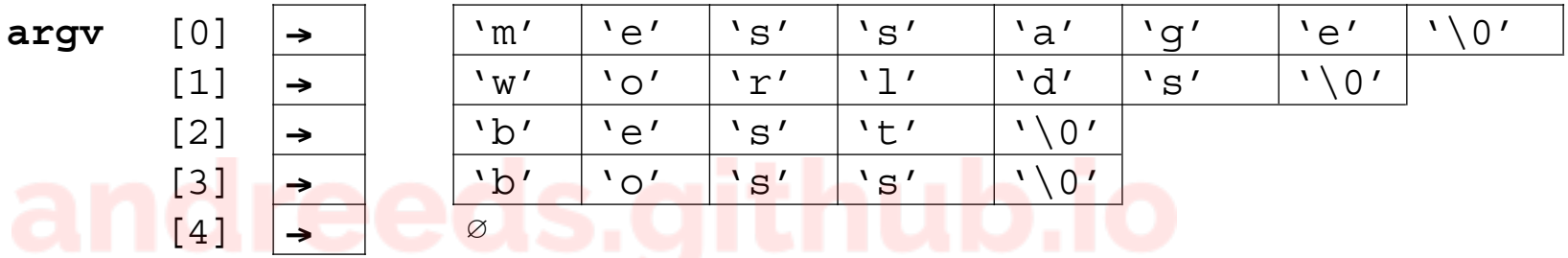
# COMMAND LINE ARGUMENTS

# ARGUMENTS TO `main()`

- It is possible to pass arguments to `main` from a command line by including parameters `argc` and `argv` in the parameter list of main
- `int main (int argc, char **argv)`
  or
- `int main (int argc, char *argv [])`
  - A command line consists of arguments separated by white space
  - An argument may not contain whitespace unless surrounded by double quotes.
  - The shell parses the command line arguments into tokens to create a parameter called `argv` consisting of an array of `argc` pointers to **`C-strings`** whose last entry is a **`NULL`** pointer

# Example

```
% message worlds best boss
```






**argc** = 4

- The first argument, **message**, is the program name
- The arguments to **message** are **worlds**, **best**, and **boss**

## commandLine.c

URCourses

```
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv [])
{
    int i = 0;
    double d;

    if (argc != 5)
    {
        fprintf (stderr, "Usage: cl arg1
arg2 arg3 9.9\n");
        return 1;
```

# THE `exec` FUNCTION

- The `exec` family of function calls provides a facility for overlaying the process image (i.e., the PCB) of the calling process with a new image from a process that is **different from that of the parent**
- An `exec` function call is used in combination with a `fork` function call
  - The `fork` function call creates a new child process
  - The `exec` function call starts a new process that overlays the child process
    - i.e., the code for the child process that resulted from the `fork` will no longer exist
  - The new process executes its code
  - The parent process continues to execute its original code

# THE exec FUNCTION

```c
#include <unistd.h>

int execl (const char *path, const char *arg0, …, char *(0));
int execv (const char *path, char *const argv []);
```

# THE `execl` PARAMETERS

- **If you know the number of command line arguments at compile time**
- The ***path*** parameter is the pathname of an executable file
- The ***file*** parameter is the name of an executable file
- The ***$arg_i$*** parameters are the individual command line arguments
- The last ***$arg_i$*** parameter should be a **NULL** pointer
  - `execl ("/bin/ls", "ls", "-l", NULL);`

## forkExecl.c

URCourses

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

int main ()
{
    pid_t childPid;
    int status;
    pid_t waitPid;

    childPid = fork ();
    if (childPid == 0)
    {
```

# THE `execv` PARAMETERS

- **If the command line arguments are determined (i.e., built) at run time**
- The ***`path`*** parameter is the pathname of an executable file
- The ***`file`*** parameter is the name of an executable file
- The ***`argv []`*** parameter is an argument array whose last element is **`NULL`**
  - `execv (argv [1], &argv [1]);`

# UNIX/LINUX PROCESSES

## forkExecv.c

URCourses

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

int main (int argc, char *argv [])
{
    pid_t childPid;
    int status;
    pid_t waitPid;

    childPid = fork ();
    if (childPid == 0)
    {
```

# THE `execvp` PARAMETERS

- **If the command line arguments are determined (i.e., built) at run time***
- **It has different behavior depending on the value of the first argument.**
  - If it contains a slash, then *file is treated as a pathname and `execvp` will behave like `execv`
  - `execvp ("/bin/ls", &argv [0]);`

or

```
char *myarg[3];
myarg[0] = strdup("ls");
myarg[1] = strdup("-l");
myarg[2] = NULL;
execvp ("/bin/ls", &myarg [0]);
```

# THE execvp PARAMETERS

or

- ```
  char *myarg[3];
  myarg[0] = strdup("ls");
  myarg[1] = strdup("-l");
  myarg[2] = NULL;
  execvp ("/bin/ls", &myarg [0]);
  ```

or

- ```
  char *myarg[4];
  myarg[0] = strdup("ls");
  myarg[1] = strdup("-l");
  myarg[2] = argv[1];
  myarg[3] = NULL;
  execvp ("/bin/ls", &myarg [0]);
  ```

## forkExecvp_1.c

URCourses

```
// Compile:
//       % gcc forkExevp_1.c -o parent
// Call:
//       % parent -l

// If
//       execvp ("/bin/ls", &argv [1]);
// Then Call
//       % parent ls -l

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

# forkExecvp_2.c

URCourses

```
// Compile:
//      gcc forkExevp_2.c -o parent
// Call:
//      % parent

#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <string.h> // because of strdup

int main (int argc, char *argv [])
{
```

# forkExecvp_3.c

URCourses

```
// Compile:
//      gcc forkExevp_3.c -o parent
// Call:
//      % parent -r
//////// -r for reversed order, you may
try another command options like -s, -S,
-i, --color

#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <string.h> // because of strdup
```