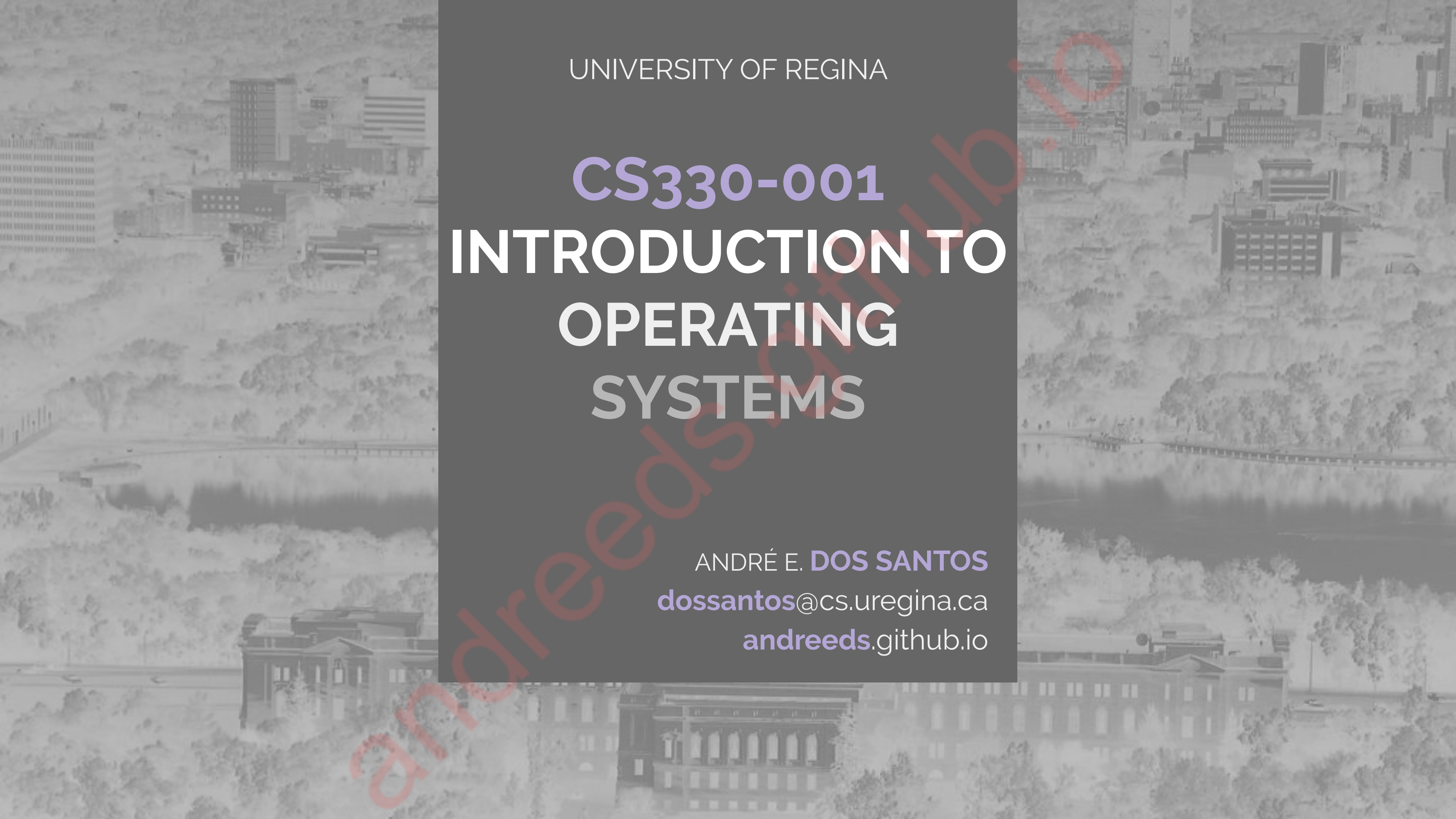


UNIVERSITY OF REGINA

CS330-001 INTRODUCTION TO OPERATING SYSTEMS

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

CS330-001
INTRODUCTION TO
OPERATING SYSTEMS

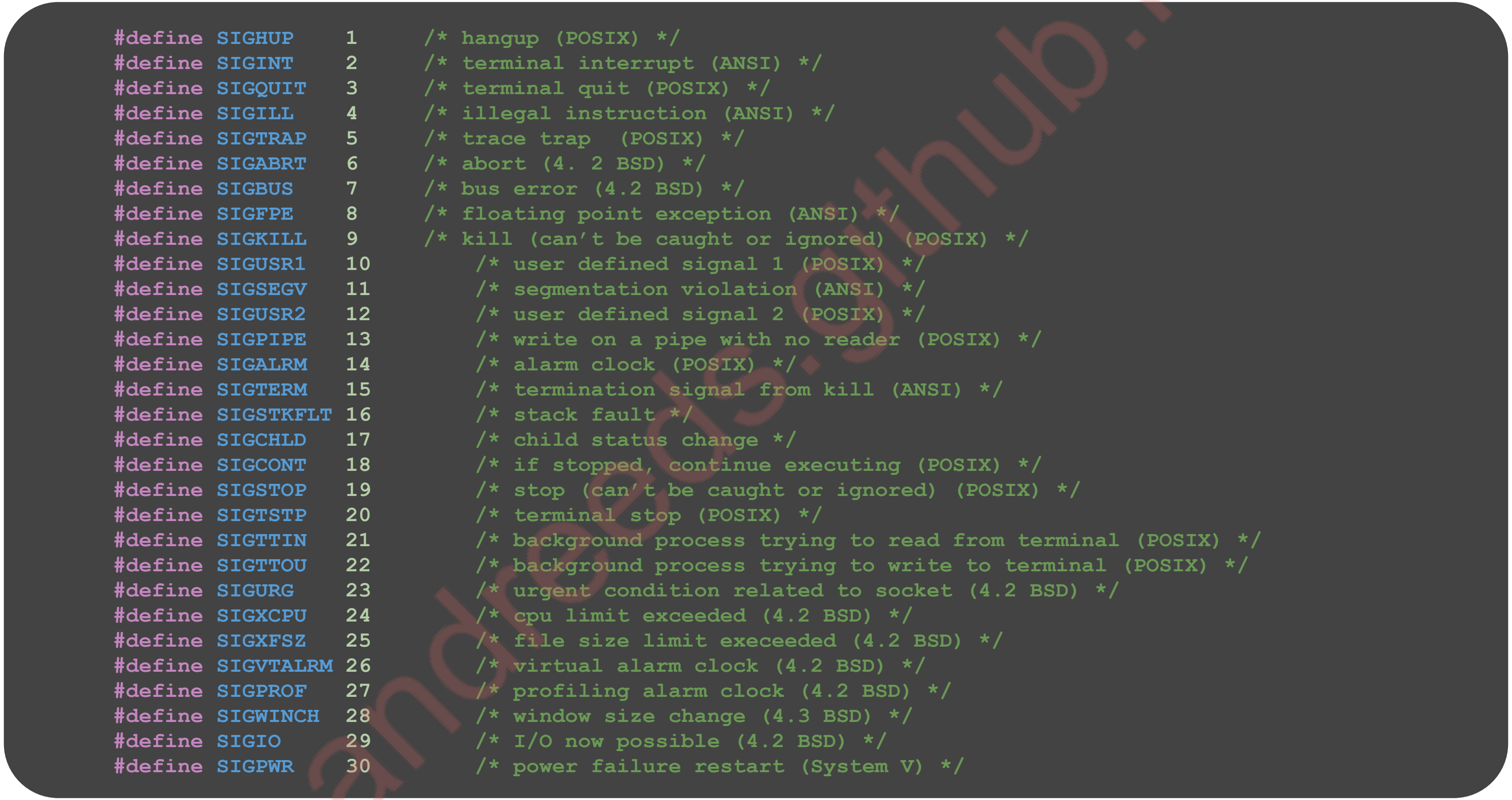
PROGRAMMING SIGNals C

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

WHAT ARE SIGNALS

- A **signal** is software interrupt mechanism that generates a notification indicating to a process that some event has occurred
- Every signal has a name and is associated with an integer-valued number

SIGNALS



```
#define SIGHUP    1     /* hangup (POSIX) */
#define SIGINT    2     /* terminal interrupt (ANSI) */
#define SIGQUIT   3     /* terminal quit (POSIX) */
#define SIGILL    4     /* illegal instruction (ANSI) */
#define SIGTRAP   5     /* trace trap  (POSIX) */
#define SIGABRT   6     /* abort (4. 2 BSD) */
#define SIGBUS    7     /* bus error (4.2 BSD) */
#define SIGFPE    8     /* floating point exception (ANSI) */
#define SIGKILL   9     /* kill (can’t be caught or ignored) (POSIX) */
#define SIGUSR1   10        /* user defined signal 1 (POSIX) */
#define SIGSEGV   11        /* segmentation violation (ANSI) */
#define SIGUSR2   12        /* user defined signal 2 (POSIX) */
#define SIGPIPE   13        /* write on a pipe with no reader (POSIX) */
#define SIGALRM   14        /* alarm clock (POSIX) */
#define SIGTERM   15        /* termination signal from kill (ANSI) */
#define SIGSTKFLT 16        /* stack fault */
#define SIGCHLD   17        /* child status change */
#define SIGCONT   18        /* if stopped, continue executing (POSIX) */
#define SIGSTOP   19        /* stop (can’t be caught or ignored) (POSIX) */
#define SIGTSTP   20        /* terminal stop (POSIX) */
#define SIGTTIN   21        /* background process trying to read from terminal (POSIX) */
#define SIGTTOU   22        /* background process trying to write to terminal (POSIX) */
#define SIGURG    23        /* urgent condition related to socket (4.2 BSD) */
#define SIGXCPU   24        /* cpu limit exceeded (4.2 BSD) */
#define SIGXFSZ   25        /* file size limit execeeded (4.2 BSD) */
#define SIGVTALRM 26        /* virtual alarm clock (4.2 BSD) */
#define SIGPROF   27        /* profiling alarm clock (4.2 BSD) */
#define SIGWINCH  28        /* window size change (4.3 BSD) */
#define SIGIO     29        /* I/O now possible (4.2 BSD) */
#define SIGPWR    30        /* power failure restart (System V) */
```

GENERATING SIGNALS

Hardware exceptions

- The conditions are detected by the hardware, which notifies the kernel, which generates the appropriate signal, which is sent to the appropriate process. Examples include:
  - Division by zero (i.e., `SIGFPE`).
  - Invalid memory reference (i.e., `SIGSEGV`).

Software conditions

- When an event happens that a process should know about. Examples include:
  - Writing to a pipe that has no reader (i.e., `SIGPIPE`).
  - When a timer set by a process expires (i.e., `SIGALRM`).
  - When some user-defined condition occurs (i.e., `SIGUSR1`).

Terminal-generated signals

- When a user presses keys simultaneously in particular combinations. Examples include:
  - Control/C to stop a runaway process (i.e., `SIGINT`).
  - Control/Z to suspend a process running in foreground (i.e., `SIGTSTP`).

GENERATING SIGNALS

- There are two generations of signals (at least for the purposes of our discussion there is):

  Unreliable

  - A throwback to the very early versions of signals in UNIX that have been superseded by the POSIX signals standard.

  Reliable

  - A (modern) version of signals adhering to the POSIX signals standard.

UNRELIABLE SIGNALS

- Unreliable signals suffer from a number of problems and should not be used in new programs:
  - They can get lost (i.e., a signal could be sent but the intended recipient misses it)
  - The disposition of a signal set by a process must be reset by the process each time the signal is received
    - If the disposition is to catch the signal (with a signal handler), but the default action is to kill the process, there is a small window of time where the default action would be enabled until the process resets it again
    - Another example of a race condition
    - They handling of a signal cannot be deferred, only ignored.

RELIABLE SIGNALS

- Reliable signals solve the problems with unreliable signals
    - The disposition of a signal set by a process is not reset to the default each time a signal is received, only when the process specifically changes it
    - Processes have the ability to both ignore or temporarily block signals
        - When a signal is blocked by a process, the kernel places it on a queue of pending signals for that process
        - A blocked signal remains pending until the process unblocks it or changes its disposition to ignore it
        - `SIGKILL` and `SIGSTOP` cannot be blocked

- From here on, we assume the use of reliable signals.

SIGNAL STATES

- A signal will always be in one of three possible states
    1. A signal is **generated** (i.e., sent to a process) when the event that causes the signal occurs.
    2. A signal is **pending** (i.e., blocked) if it has been generated but not delivered.
    3. A signal is **delivered** when the action associated with the signal is actually invoked

- The **lifetime** of a signal is the interval between its generation and delivery.

SIGNAL GENERATION

- Signals may be generated in two ways:
  1. **Synchronously**: When an event occurs that is directly caused by the execution of a process' code (also called a trap) (e.g., `SIGFPE`)
  2. **Asynchronously**: When an event occurs at a seemingly random time with respect to the process (e.g., `SIGKILL`)

SIGNAL RESPONSE

A process can respond to the receipt of a signal (called the **signal's disposition** or **associated action**) in two ways when it is delivered:

- **Catch it:** Call a signal handler, a user-written function contained in a process that describes how the event should be handled
  - Examples include:
    - Catching `SIGTERM` (the default termination signal sent by the `kill` command) to release memory and delete temporary files
    - Catching `SIGCHLD` to catch the termination of a child process
- **Take one of five possible default actions:**
  - Ignore the signal
  - Terminate the process
  - Core dump
  - Stop if the process is currently running
  - Continue if the process is currently stopped

`kill` COMMAND

- List the symbolic names of the signals available (POSIX)

```
kill -l
```

- Kill a **particular** process (POSIX)

```
kill -s signal_name pid
```

- Traditional kill command (still supported by POSIX, but only because of widespread usage)

```
kill -signal_name pid
kill -signal_number pid
```

ELEMENTARY SIGNAL SYSTEM CALLS

The `kill` system call is used to send a signal to a process

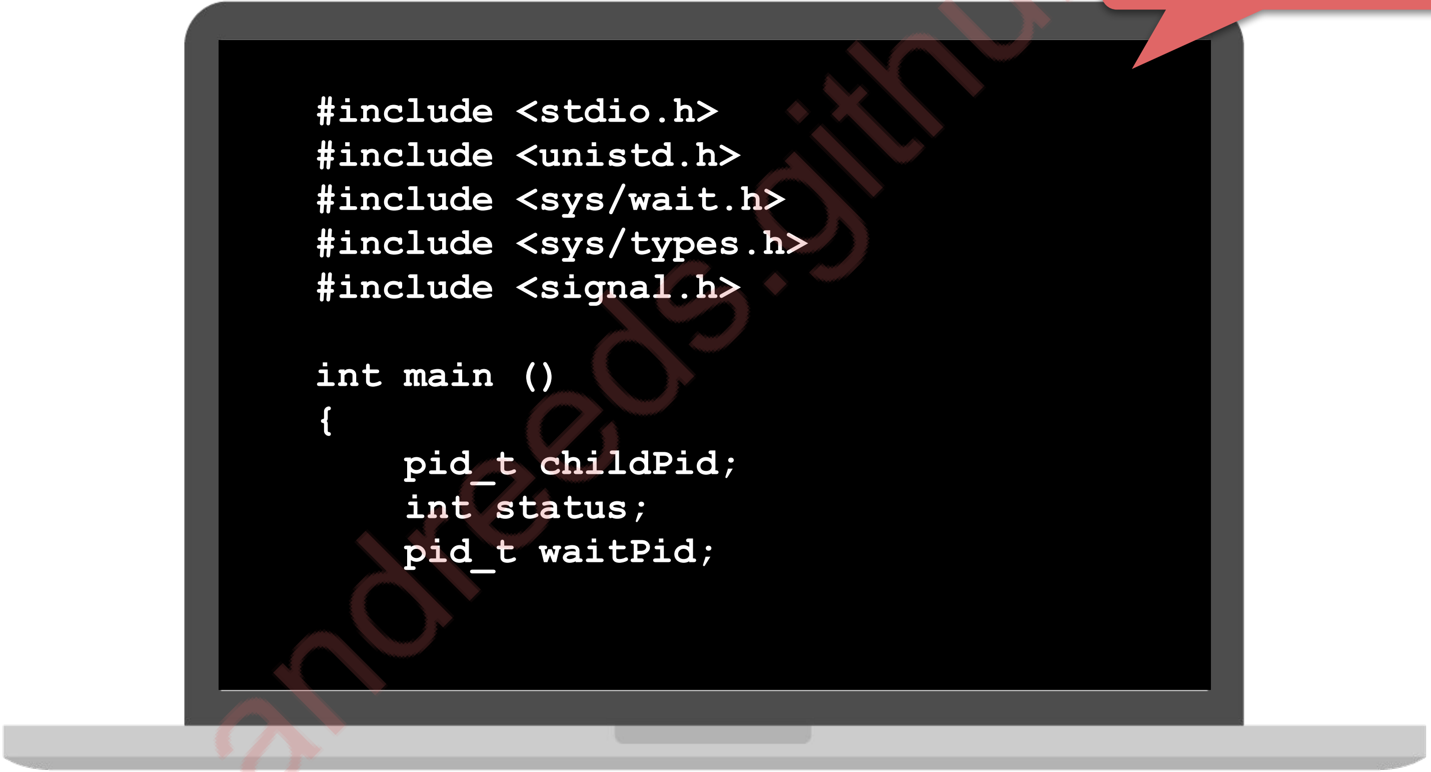
```
#include <sys/types.h>
#include <signal.h>
int kill (pid_t pid, int sig);
```

- The `kill` system call sends the signal specified by `sig` to the process specified by `pid`
- `pid` is a valid process identifier
- `sig` must be a valid signal name or **0**
  - If sig is 0, (i.e., the NULL signal), normal error checking is performed, but no signal is actually sent
  - Why would we want to do this?
    - We can use 0 to check whether pid is a valid process before we actually try to kill it
- If **successful**, kill returns **0**
- If **unsuccessful**, kill returns **–1** and sets `errno`

ELEMENTARY SIGNAL SYSTEM CALLS

childKillsParent.c

URCourses



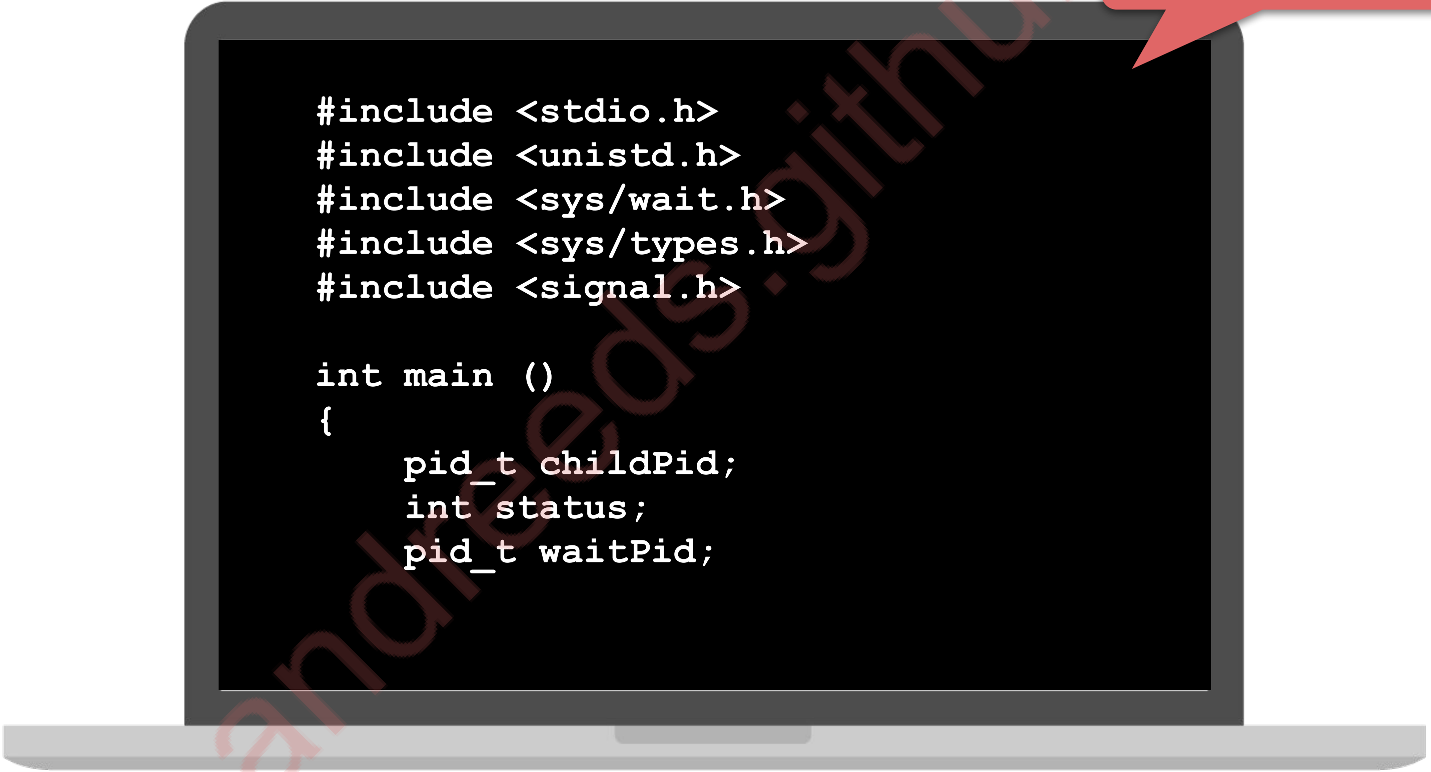
```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <signal.h>

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

ELEMENTARY SIGNAL SYSTEM CALLS

catchSignals.c

URCourses



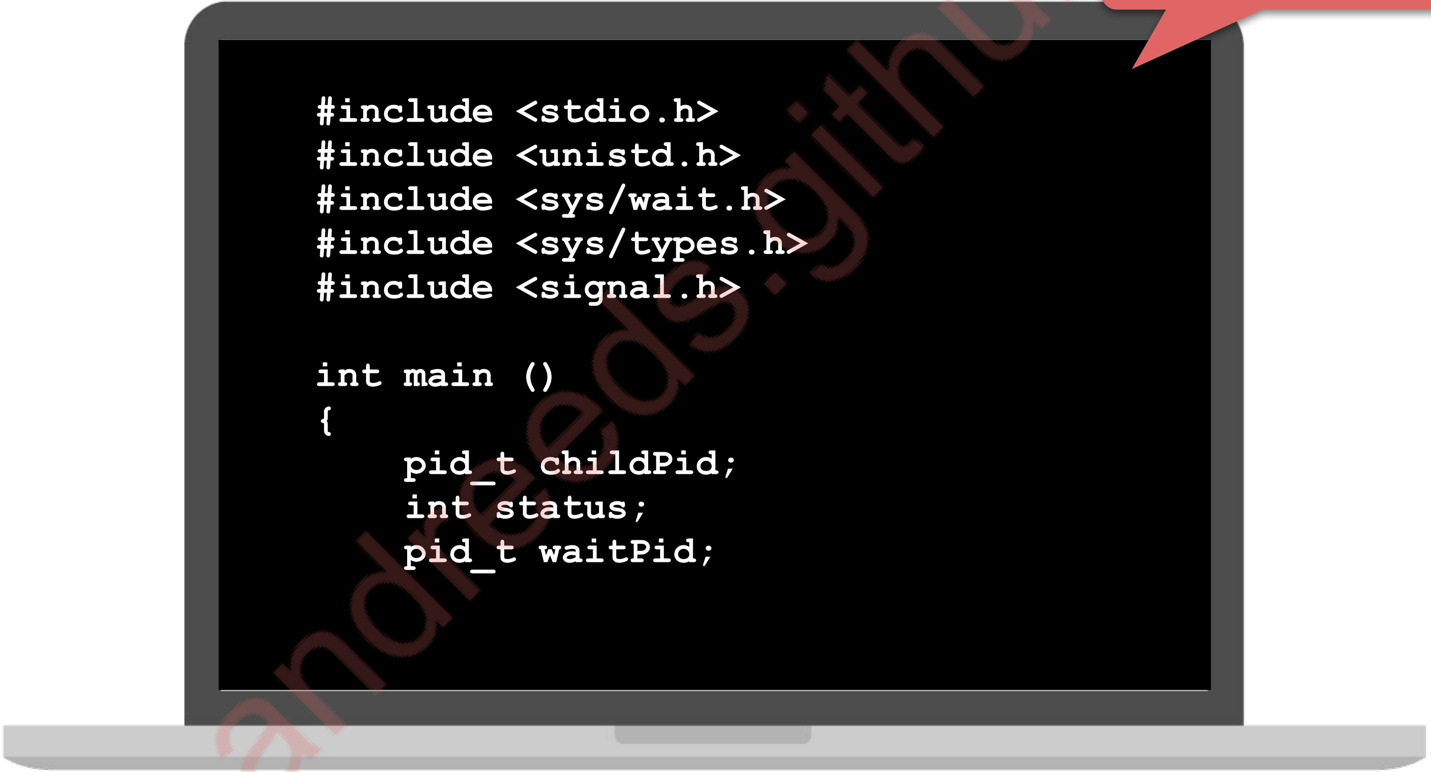
```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <signal.h>

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

ELEMENTARY SIGNAL SYSTEM CALLS

parentCatchSignals.c

URCourses



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

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