Assignment #1

- You will create a program that uses several system calls demonstrating various process attributes and behavior
- During execution, your program will create a child process that will eventually begin to run a program you will find at:

  ~bill/cs308/Assign1

- The source code for this program is at:

  http://www.cs.uml.edu/~bill/cs308/Assign1.c

  and at:

  ~bill/cs308/Assign1.c

Using System Calls

- The lowest level access to kernel support is exported to user space in the form of a set of functions called ***system calls***
- The Linux system call APIs are wrapped in a collection of routines from the ***libc (or glibc)*** platform library
- The libc library is linked by default when using ***gcc*** to build an executable program

Using System Calls (cont'd)

- Most of the specific libc system call functions have a simple return type of ***int***
- A system call may return any appropriate integer value, but a return value of ***-1*** is typically an indication that the system call failed for some reason
- The linker supplied reference to ***int errno*** can then be used to determine what the failing error was
- The libc library has a routine called ***perror()*** which can look-up and print the error message that corresponds to a given errno

Using System Calls (cont'd)

- If you use system calls that can fail in your programs for this course, I expect you to use them with the general format:

```
if ( (int x = system_call()) == -1 ) {
      perror ("my message ");
      exit(n);  // non-zero value, < 256
} // end if
```

Using System Calls (cont'd)

- In come cases we may be interested in a specific non-failed system call return, leading to a second format:

```
switch ( int x = system_call()) {
  case -1:
       perror (“my message “);
       exit(n);
  case some_val:
       // some action
  default:
       // some action
} // end switch
```

Using System Calls (cont'd)

- An example of the second format:

```
switch( int x = fork() ){
  case -1: // no child was created
     perror("fork call failed ");
     exit(1);
  case 0:  // only child can get here
     printf("The child is alive \n");
     // additional child actions
  defult:  // only parent can get here
     printf("Parent created PID %d \n", x);
     // additional parent actions
} // end switch
```

Your Program Should Have the Following Shape

```
void    sigfunc (int signum){
        // your signal handler function, includes execl call
}

int     main (void){
        // create pipe with pipe call
        // print out your credentials
        // fork child, block on pipe with read call
                // child installs sigfunc with sigaction call
                // child must print out credentials
                // child must write pipe with write call
                // child enters endless loop
        // parent wakes from pipe read (after child writes)
        // parent sends SIGTERM to child pid with kill call
        // parent blocks on wait call
                // child moves to sigfunc when SIGTERM arrives
                // sigfunc must load prof program with execl
                // prof program will print out credentials
                // prof program installs its own signal handler
                // prof program prompts user for kill command
                // prof program enters endless loop
                        // user enters shell kill command
                // prof program enters signal handler, will exit
        // parent wakes from wait call when child dies
        // parent prints child term status and finishes
}
```

Assignment #1 Procedures

- Create a program that will:
  - Declare a typedef for the exit status information returned from the **wait()** call (**pid = wait(int *status)**)

```
typedef union{
        int exit_status;
        struct{
                unsigned  sig_ num:7;
                unsigned  core_dmp:1;
                unsigned  exit_num:8;
        }parts;
}LE_Wait_Status;
```

  - Include a signal handler function to be inherited by a child process (this function will load the Assign1 executable)
  - Create a pipe using the **pipe()** system call to be inherited by a child process
  - Print out its own credentials as shown in the on-line source code for Assign1.c

Assignment #1 Procedures (cont'd)

- Your program will then:
  - Create a child process using the **`fork()`** call
  - The parent will then read the read-channel of the pipe with the **`read()`** system call, blocking until the child writes the pipe with the **`write()`** system call
  - When the parent awakes from the pipe read it will send signal SIGTERM (#15) using the **`kill()`** system call to the child process (the child will then be in its endless loop)
  - The parent now awaits the death of the child in the **`wait()`** system call
  - When the child dies, the parent prints out the child's exit status and then terminates (returns) itself

Assignment #1 Procedures (cont'd)

- The child process that you create will:
  - Come into existence on the return side of the **`fork()`** call that the parent made
  - Set up the signal handler function using the **`sigaction()`** system call to catch the signal the parent will send (SIGTERM)
  - Collect and print out its credentials as did the parent
  - Write a character into the pipe using the **`write()`** call to wake up the parent after its credentials are printed
  - Enter an endless loop, expecting the arrival of the parent signal to force it to execute the signal handler function (there should be a timeout in this loop to terminate in the event the signal doesn't arrive)

Assignment #1 Procedures (cont'd)

- The child, upon catching the parent's signal, will:
  - Use the **`execl()`** call to load in the executable built from the source code Assign1.c (this executable can be found on mercury at: ~bill/cs308/Assign1 or you can just build one yourself from the Assign1.c source code found on-line, and also at ~bill/cs308/Assign1.c)
  - When the Assign1.c program is loaded into the child process, it will collect and print its credentials and also print a message asking the user to enter the kill command from the shell command line
  - It is important that you start your main program as a **background process**, so that you can use the shell command line now to enter the required kill command

Assignment #1 Procedures (cont'd)

- The Assign1.c program will then:
  - Enter an endless loop, expecting the SIGTERM signal to arrive when the user types in the kill command
  - When this program gets the SIGTERM signal it will run its handler which will call **`exit()`** to terminate
  - When the child terminates, the parent will awake from its **`wait()`** call and report how the child terminated (exit or signal, core dump or no-core dump)
  - The parent program will then do a normal termination (return) after printing the child's termination status

Assignment #1 Procedures (cont'd)

- System calls you will need (see the man pages):
  - **getpid ()**
  - **getppid ()**
  - **getuid ()**
  - **geteuid ()**
  - **getgid ()**
  - **getegid ()**
  - **getpriority ()**
  - **read ()**
  - **write ()**
  - **sigprocmask ()**
  - **sigaction ()**
  - **fork ()**
  - **execl ()**
  - **kill ()**
  - **wait ()**
  - **pipe ()**
  - **exit ()**