CS 3410 Lab 9

Cornell Bowers C·IS
Computer Science

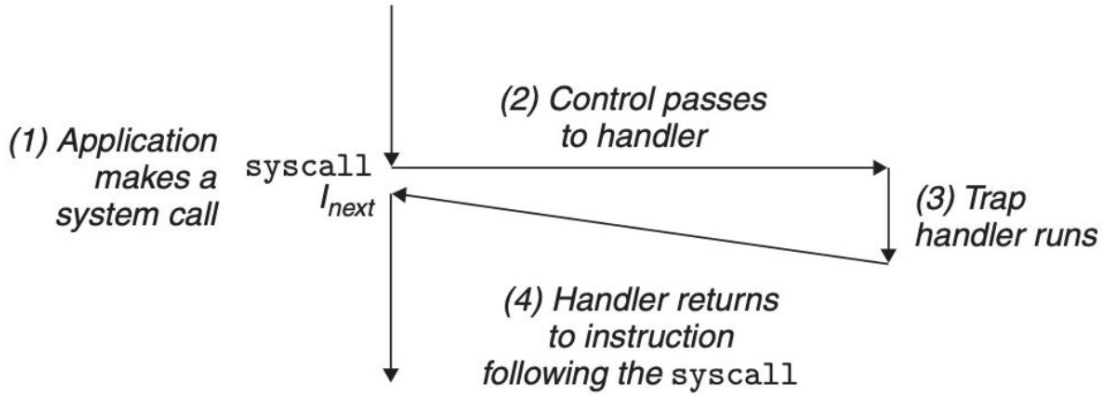
Agenda

1. Review
2. System Calls
3. A9 Tips

Review

System Calls

- Procedure-like interface between user programs and the kernel
  - User programs may need to request services from the kernel such as reading a file (**`read`**) or loading a new program (**`execve`**)
- Processors provide an ecall instruction that causes a trap to the exception handler and and calls the appropriate kernel routine



Error Handling

- Return `-1` and set the global integer variable **`errno`** with error message.
- Linux **`man pages`** contains documentation on system libraries and system calls.
- Programmers should always check for errors when using system calls:

```
int main(int argc, char** argv) {
  pid_t pid = fork();
  if (pid < 0) {
    fprintf(stderr, "fork error: %s\n", strerror(errno));
    exit(-1);
  }
}
```

Interrupting a System Call

- What if a signal occurs when a long-running system call is being run (i.e. waitpid)?
- Operating System may stop the system call with an error and indicate the `errrno == EINTR` code
- Many libraries will restart the system call in such cases, including glibc whenever printf is called and many Python functions
- It is rare to encounter this error code on modern systems, but it is still helpful to catch it

Pipes

- How do we communicate between processes (including sending data to stdin)?
- pipe() creates a pipe, a unidirectional data channel that can be used for interprocess communication
- Pipes provide you with an array, pipefd used to return two file descriptors referring to the ends of the pipe
- pipefd[0] refers to the read end of the pipe
- pipefd[1] refers to the write end of the pipe

Implementation and Worksheet

- Visit the lab page and work through the examples
- Complete question 1 in the worksheet

System calls

Process IDs

- Each process has a unique positive (nonzero) **process ID (PID)**
  - The `getpid` function returns the PID of the *calling* process
  - The `getppid` function returns the PID of the *parent* process
- Both functions return a value of type `pid_t`, which on Linux systems is defined as `int`

Fork

- A *parent process* creates a new running *child process* by calling the `fork` function
- The newly created child process is nearly identical to the parent process: the child gets an identical copy of the parent's user-level virtual address space
  - The child also gets identical copies of any of the parent's open file descriptors
- The most significant difference is that the parent and child have different PIDs
- `fork` is called once and returns twice: once in the parent and once in the child

Fork

- **Duplicate but separate address spaces**: x has different values in the parent and the child
- **Shared files**: both the parent and child print their output because the child inherits stdout from the parent

```
int main(int argc, char** argv) {
  int x = 1;
  pid_t pid = fork();
  if (pid < 0) { unix_error("fork error"); }

  if (!pid) {
    printf("child: x = %d\n", ++x);
    exit(0);
  }

  printf("parent: x = %d\n", --x);
  exit(0);
}
```

Signals

- A **signal** is a small message that notifies a process that an event of some type has occurred in the system
- Each signal type corresponds to some kind of system event
- Unix systems provide mechanisms for sending signals to processes through **process groups**
  - Every process belongs to exactly one **process group**, which is identified by a **process group ID**
  - The `getpgrp` function returns the process group ID of the current process
  - `setpgid` changes the process group of a process
- Processes send signals to other processes (including themselves) by calling the `kill` function

Worksheet

Complete parts 2 and 3

Reaping Child Processes

- When a process terminates the kernel does not immediately remove it from the system
  - The process is kept around in a terminated state until it is **reaped** by the parent
  - A terminated process that has not yet been reaped is called a zombie
- A parent waits for its children to terminate or stop by calling the `waitpid` function

Reaping Child Processes

Optional `options` include `WNOHANG` and `WUNTRACED`:

- `WNOHANG`: returns 0 immediately if no child processes stopped/terminated
- `WUNTRACED`: waits until child stops or terminates
- `WNOHANG | WUNTRACED`:
  - If no child processes stopped/terminated, returns 0 immediately
  - Otherwise, returns PID of one of the stopped or terminated children.

Checking the Exit Status of a Reaped Child

These functions accept `status` as their only argument:

- `WEXITSTATUS`: the exit status of child
- `WIFEXITED` checks if child is terminated
- `WIFSIGNALED` checks if child process exited with an uncaught signal
- `WIFSTOPPED` checks if child is currently stopped
- `WTERMSIG`: signal (as an `int`) that caused child process to terminate
- `WSTOPSIG`: signal (as an `int`) that caused child to stop

Worksheet

Complete part 4

Assignment Tips

A9 Tips

- **Read the entire write up** before starting
- The majority of your time in this assignment will not be spent on the common case, but rather error handling and resource cleanup
- The assignment write up contains many useful links to the Linux Manual Pages: https://linux.die.net/man/
- Read each entry carefully and decide if it can help you!
- For other functions, use the linux command `man 2 <system call>` or `man 3 <library call>`

Good Luck