

CS 3410 Lab 10

Fall 2025



Agenda

1 Control Flow

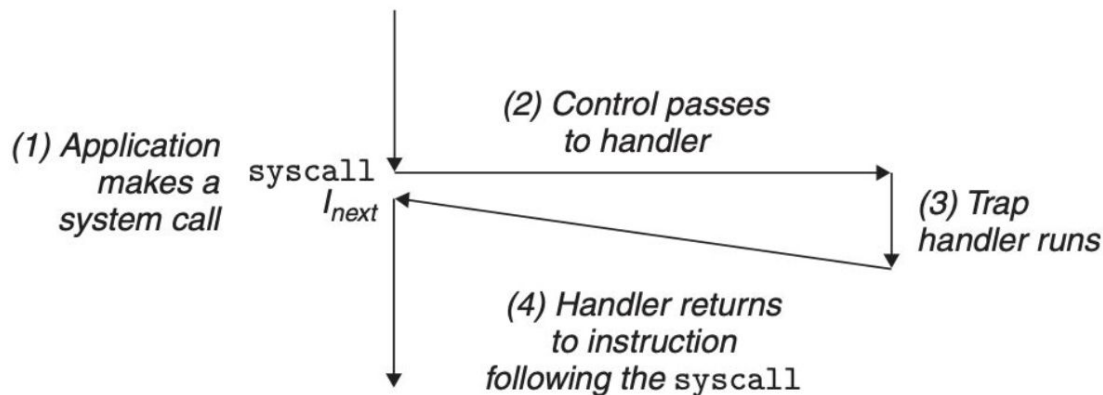
2 System Calls

3 A10 Tips

Control Flow

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 a **syscall** instruction that causes a trap to the exception handler 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);  
    }  
}
```



Exceptions and Interrupts

- What if program runs out of memory/executes an illegal instruction?
- The same hardware mechanism that handles system calls also handles exceptions and interrupts!
- Interrupts are critical for proper program function e.g. mouse movement, key strokes.
- Instead of busy waiting on a disk read, OS can interrupt the program when its ready.



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

A10 Tips

- The assignment write up has many useful links to the Linux Manual Pages
- Linux Manual Pages are your best friend: <https://linux.die.net/man/>
- Read each entry carefully and decide if it can help you!



Good Luck!