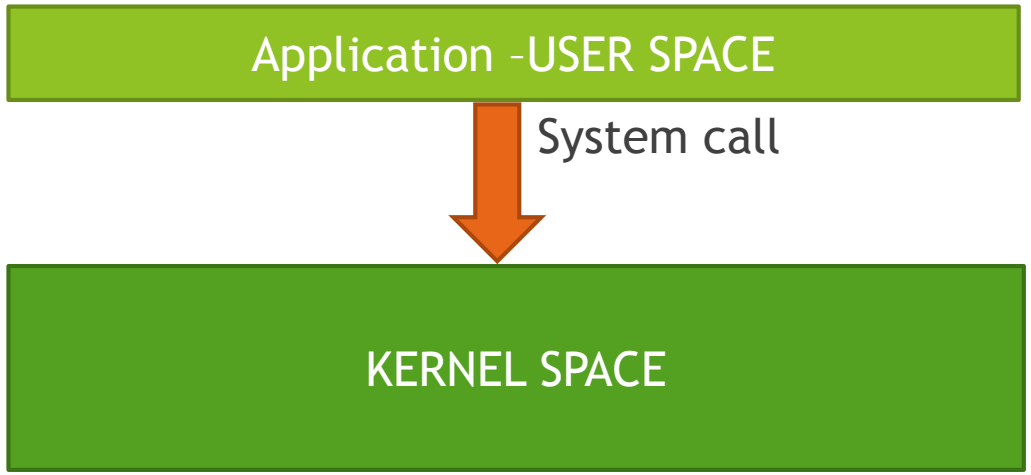
System Calls

- If a **process** is running a user program in user mode and needs a system service, such as reading data from a file, it has to execute a **trap instruction** to transfer control the **operating system**.



System Calls

A system call is a request for service that a program makes of the kernel. The service is generally something that only the kernel has the privilege to do, such as doing I/O.

SYSTEM CALLS

PROCESS CONTROL	fork(), wait(), exec(),exit()
FILE MANIPULATION	open(), close(), read(), write()
DIRECTORIES MANAGEMENT	mkdir(),rmdir(), mount(),link()
OTHER	chdir(),chmod(), kill(),time()

Fork()

- Fork creates a new process(**child process**).
  - It creates an exact duplicate of the original process, including all the file descriptors, registers etc.
- **The fork is called once, but returns twice!**
  - After the fork, the original process and the copy(the parent and the child) go their separate ways.
  - The fork call returns a value, which is zero in the child and equal to the child's process identifier (**PID)** in the parent.
- Now consider how fork is used by the shell. When a command is typed, the shell forks off a new process. This child process must execute the user command.

Fork() – PID (Process IDentity)

- **pid < 0** → the creation of a child process was unsuccessful.
- **pid == 0** → the newly created child.
- **pid > 0** → the process ID of the child process passes to the parent

Consider the program:

```
#include <unistd.h>

pid_t pid = fork();
printf("PID:%d\n",pid);
…
```

The parent will print:
PID:34
The child will always print:
PID:0

Fork()

```
#define TRUE 1
while (TRUE) {                                  /* repeat forever */
   type_prompt();                               /* display prompt on the screen */
   read_command(command, parameters);           /* read input from terminal */
   if (fork() != 0) {                           /* fork off child process */
      /* Parent code. */
      waitpid(-1, &status, 0);                  /* wait for child to exit */
   } else {
      /* Child code. */
      execve(command, parameters, 0);           /* execute command */
   }
}
```

Exec (binary_path)

- The exec() call **replaces/overwrites** a current process image with a new one (i.e. loads a new program within the current process).
- The file descriptor table remains the same as the original process.
- Argument passed via exec() appear in the argv[] of the main function.
- Upon success, exec() **never** returns to the caller.
  - It replaces the current process image, so it cannot return anything to the program that made the call.
  - If it does return, it means the call failed

exec("/bin/ls"): overwrites the memory code image with binary from /bin/ls and execute.

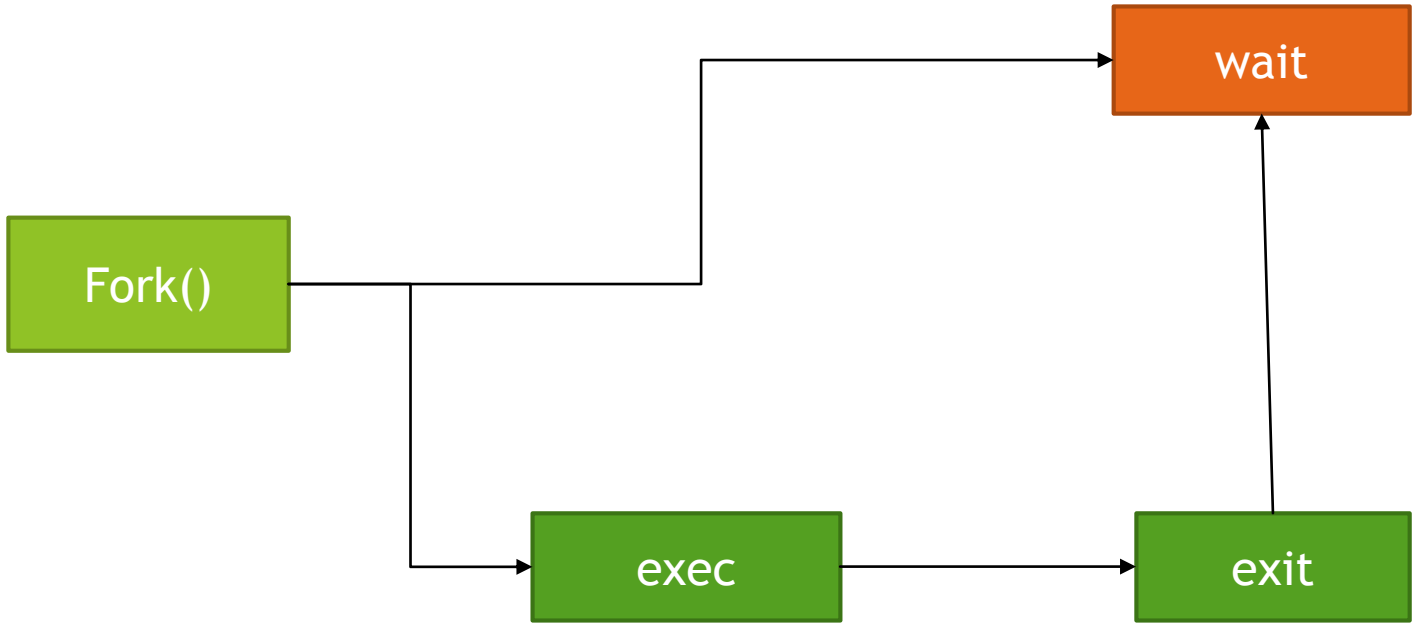
Exec(binary_path)

- There's not a syscall under the same exec().
- By **exec()** we usually refer to a family of calls:
  - int execl(char *path, char *arg, ...);
  - int execv(char *path, char *argv[]);
  - int execle(char *path, char *arg, ..., char *envp[]);
  - int execve(char *path, char *argv[], char *envp[]);
  - int execlp(char *file, char *arg, ...);
  - int execvp(char *file, char *argv[]);

Where: **l**=argument list, **v**= argument vector, **e**=environmental vector, **p**= search path

Fork and exec

- Often after doing a fork() we want to load a new program into the child. E.g.: a shell



Wait()

- Forces the parent to **suspend** execution, i.e. wait for its children or a specific child to die(terminate).
- When the child process dies, it returns an exit status to the OS, which is then returned to the waiting parent process. The parent process then resumes execution.
- A child process that dies but is never waited on by its parent becomes a **zombie process**. Such a process continues to exist as entry in the system process table even though it is no longer an actively executing program.

Exit()

- This call **gracefully** terminates process execution. Gracefully means it does clean up and release of resources, and puts the process into the **zombie state.**
- By calling wait(), the parent cleans up all its zombie children.
- When the child process dies, an exit status is returned to the OS and a signal is sent to the parent process.
- The exit status can then be retrieved by the parent process via the **wait** system call.

Fork, exec and wait

```
while (1) {                                  /* repeat forever */
   type_prompt();                            /* display prompt on the screen */
   read_command(command, parameters);        /* read input from terminal */
   if (fork()!= 0) {                         /* fork off child process */
      /* Parent code. */
      waitpid(-1, &status, 0);               /* wait for child to exit */
   } else {
      /* Child code. */
      execve(command, parameters, 0);        /* execute command */
   }
}
```

State of a process

In computing, a process is an instance of a computer program that is being executed. It contains the program code and its current activity.

- **Orphan process,** is a computer process whose parent process has finished or terminated, though it remains running itself.
- **Daemon process,** runs as a background process, rather than being under the direct control of an interactive user.
- **Zombie process,** is a process that has completed execution but still has an entry in the process table.

Pipes

- Pipes provide a unidirectional interprocess communication channel.
- “|” (pipe) operator between two command directs the stdout of the first to the stdin of the second. Any of the commands may have options or arguments.

- E.g. of pipelines:
  - Command 1| command 2 parameter 1
  - ls –l | grep key

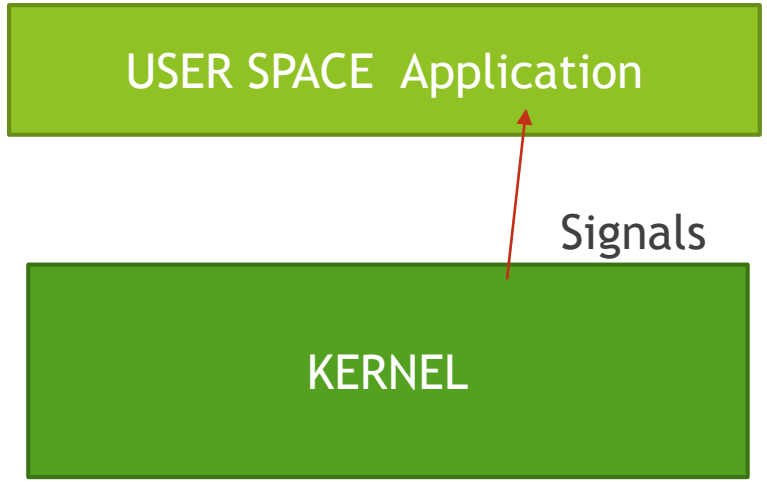
```
void main(int argc, char *argv[]){
   int pipefd[2];
   pid_t cpid;
   char buf;
   if (pipe(pipefd) == -1) {
      perror("pipe");
      exit(EXIT_FAILURE); }
      cpid = fork();
      if (cpid == -1) {
         perror("fork");
         exit(EXIT_FAILURE); }
         if (cpid == 0) {      /* Child reads from pipe */
            close(pipefd[1]);   /* Close unused write end */
            while (read(pipefd[0], &buf, 1) > 0)
               write(STDOUT_FILENO, &buf, 1);
               write(STDOUT_FILENO, "\n", 1);
               close(pipefd[0]);
               exit(EXIT_SUCCESS);
         } else {       /* Parent writes argv[1] to pipe */
            close(pipefd[0]);     /* Close unused read end */
            write(pipefd[1], argv[1], strlen(argv[1]));
            close(pipefd[1]);/* Reader will see EOF */
            wait(NULL);    /* Wait for child */
            exit(EXIT_SUCCESS); }
}
```

Signals

- A signal is an asynchronous event which is delivered to a process.
- Asynchronous means that the event can occur at any time
  - May be unrelated to the execution of the process
  - E.g. user types ctrl-C, or the modem hangs
- Unix supports a signal facility, looks like a software version of the interrupt subsystem of the normal CPU
- Process can send a signal to another – Kernel can send signal to a process
- A process can:
  - Ignore/discard the signal (not possible with SIGKILL or SIGSTOP)
  - Execute a signal handler function, and then possibly resume execution or terminate
  - Carry out the default action for that signal

Signals

The signal() system call installs a new signal handler for the signal with the number signum. The signal handler is set to sighandler which may be a user specified function.



Example

```
int main() {
   signal(SIGINT,foo); …
   return 0; }
void foo(int signo) {
      … /*deal with SIGINT*/
      return; }
```

Flow control

- Flow control is to prevent too fast of a flow of bytes from overrunning a terminal.
- Software flow control is a method of flow control. It uses special codes call XOFF and XON(from “transmit off” and “transmit on”).

Code	Meaning	Keyboard
XOFF	Pause transmission	Ctrl+S
XON	Resume transmission	Ctrl+Q

Redirection

- Use **dup2**()
  - dup2(source_fd, destination_fd)
- **Standard Input “<“**
  - E.g. sort < file_list.txt
- **Standard Output “>”,”>>”**
  - e.g. ls > file_list.txt
  - e.g. ls >> file_list.txt (append)
- Use **fopen()**
  - **“r”** for input “<“
  - **“w+”** for output “>”
  - **“a”** for append output “>>”

Assignment 1

A C shell (command interpreter) that reads user commands and executes them.

- getlogin() (if does not work try: struct passwd *pw = getpwuid(getuid());
  printf("username:%s\n",pw->pw_name);)
- Implement character flow control (see termios)
- Simple commands such as:
  - cd (see chdir)
  - fg (brings a process from background in the foreground)
  - exit
  - Also
    - ls, ls -l, ls -a -l, cat file.txt, sort -r -o output.txt file_to_sort.txt, …

Assignment 1

A C shell (command interpreter) that reads user commands and executes them.

- User can send a signal by pressing Ctrl-z to put a foreground process in the background.
- Complex commands such as:
  - Redirection of input and output (see dup2())
    - ls -l > output
    - cat < input
    - cat < input > output
  - Pipes (see pipe())
    - ps axl | grep zombie
    - ps axl | grep zombie > output
    - ls | grep “.c”

Assignment 1

1. Print prompt
2. Read command
    1. Parse command and look for “-,|,>,>>,<,&”

    If command == exit terminate shell

    Else if command == cd use chdir

    Else if command == fg bring in the foreground the background process

    2.2 fork

    parent

    if command has “&” work in the background else wait

    child

    If command has “|” use pipe

    If command has “>,>>,<“ use dup2()

    Exec()

    Go back to step 1

Useful links

- https://linux.die.net/man/3/exec
- https://linux.die.net/man/2/fork
- https://linux.die.net/man/2/wait
- https://linux.die.net/man/2/pipe
- https://linux.die.net/man/2/dup2
- https://www.tutorialspoint.com/c_standard_library/c_function_fopen.htm
- http://man7.org/linux/man-pages/man2/pipe.2.html
- http://man7.org/linux/man-pages/man3/termios.3.html