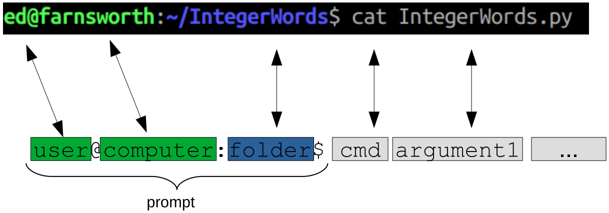
What is a Shell?

- A program!
- A mechanism to interact with the computer / OS directly
- The CLI is just one type of shell



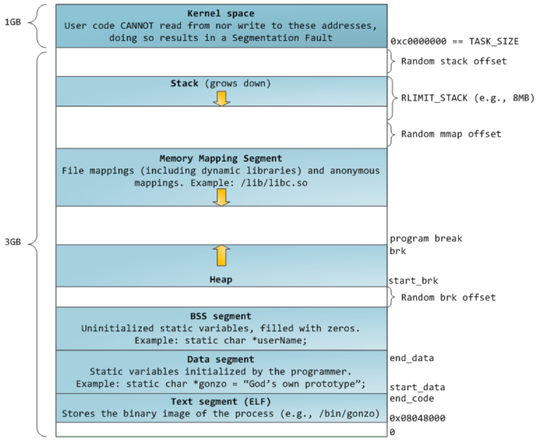
```
ed@farnsworth:~/IntegerWords$ cat IntegerWords.py
```

Shell Psuedocode

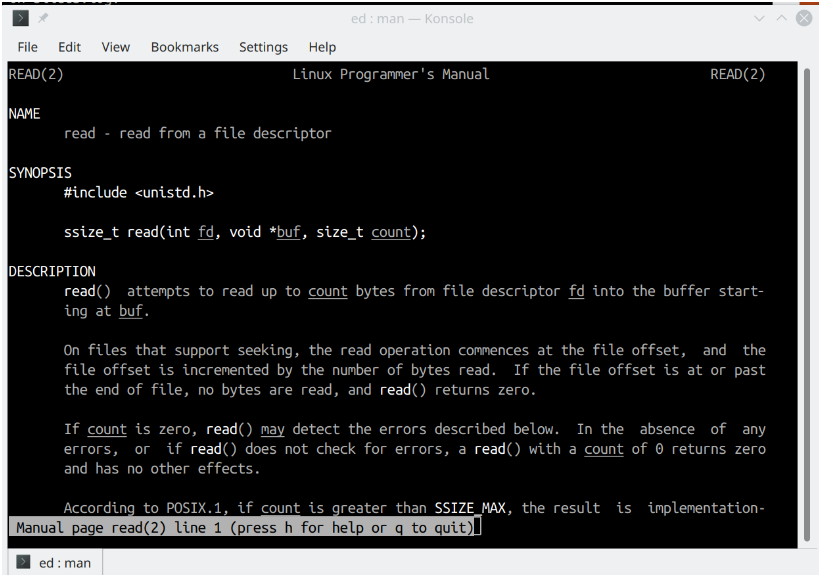
```
while(1){
    printf("person@machine$ ")

    // read_command();
    // cmd = command parsed
    // params = parameters parsed

    if(is_built_in(cmd))
    {
        do_built_in(cmd);
    }

    else
    {
        if(fork() != 0){
            waitpid(...?); // Parent
        } else {
            execve(cmd, params, 0); // Child
        }
    }
}
```

What Is A System Call?

- Functions provided by OS/kernel
- Run in kernel space
- Do important work for processes
- An interface to the machine


1GB
3GB
Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault
0xc0000000 == TASK_SIZE
Random stack offset
Stack (grows down)
RLIMIT_STACK (e.g., 8MB)
Random mmap offset
Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so
program break
brk
Heap
start_brk
Random brk offset
BSS segment
Uninitialized static variables, filled with zeros. Example: static char *userName;
end_data
Data segment
Static variables initialized by the programmer. Example: static char *gonzo = "God's own prototype";
start_data
end_code
Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)
0x08048000
0

read Man Page

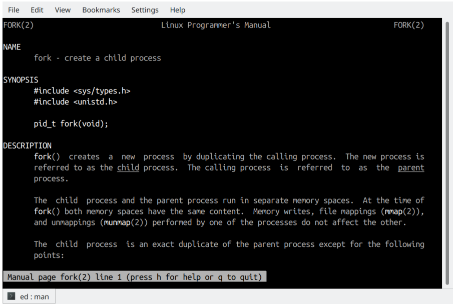


```
READ(2)                    Linux Programmer's Manual                   READ(2)

NAME
       read - read from a file descriptor

SYNOPSIS
       #include <unistd.h>

       ssize_t read(int fd, void *buf, size_t count);

DESCRIPTION
       read()  attempts to read up to count bytes from file descriptor fd into the buffer start‐
       ing at buf.

       On files that support seeking, the read operation commences at the file offset,  and  the
       file offset is incremented by the number of bytes read.  If the file offset is at or past
       the end of file, no bytes are read, and read() returns zero.

       If count is zero, read() may detect the errors described below.  In the  absence  of  any
       errors,  or  if read() does not check for errors, a read() with a count of 0 returns zero
       and has no other effects.

       According to POSIX.1, if count is greater than SSIZE_MAX, the result  is  implementation-
 Manual page read(2) line 1 (press h for help or q to quit)
```

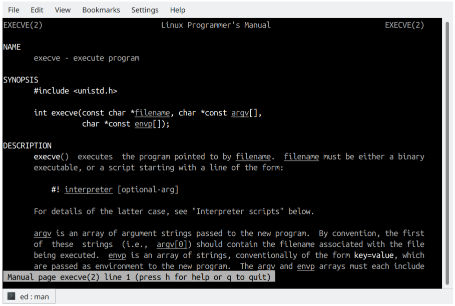
Example System Calls

- chown
- chdir
- open
- read
- write
- close
- fork
- execve
- exit
- ioctl
- lseek
- mkdir
- mmap
- pipe
- reboot
- shutdown
- ...

Man Pages

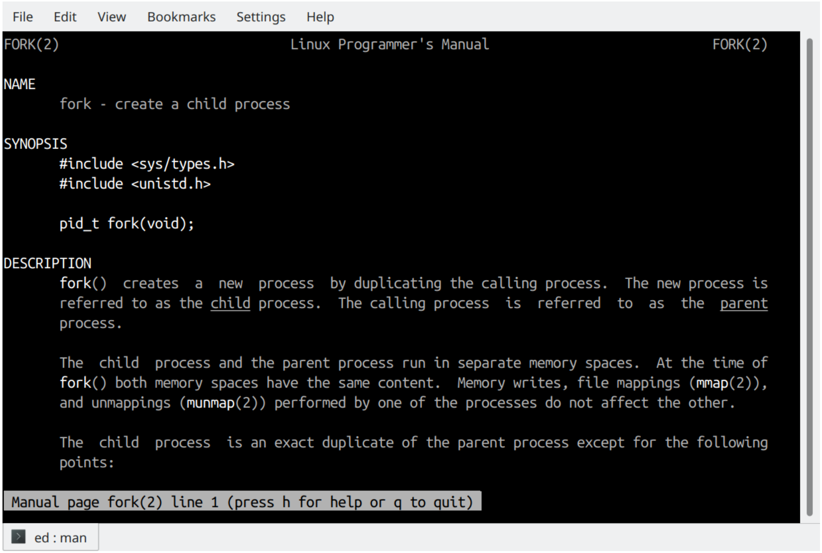


```
FORK(2)                    Linux Programmer's Manual                   FORK(2)

NAME
       fork - create a child process

SYNOPSIS
       #include <sys/types.h>
       #include <unistd.h>

       pid_t fork(void);

DESCRIPTION
       fork()  creates  a  new  process  by duplicating the calling process.  The new process is
       referred to as the child process.  The calling process  is  referred  to  as  the  parent
       process.

       The  child  process and the parent process run in separate memory spaces.  At the time of
       fork() both memory spaces have the same content.  Memory writes, file mappings (mmap(2)),
       and unmappings (munmap(2)) performed by one of the processes do not affect the other.

       The  child  process  is an exact duplicate of the parent process except for the following
       points:

 Manual page fork(2) line 1 (press h for help or q to quit)
```



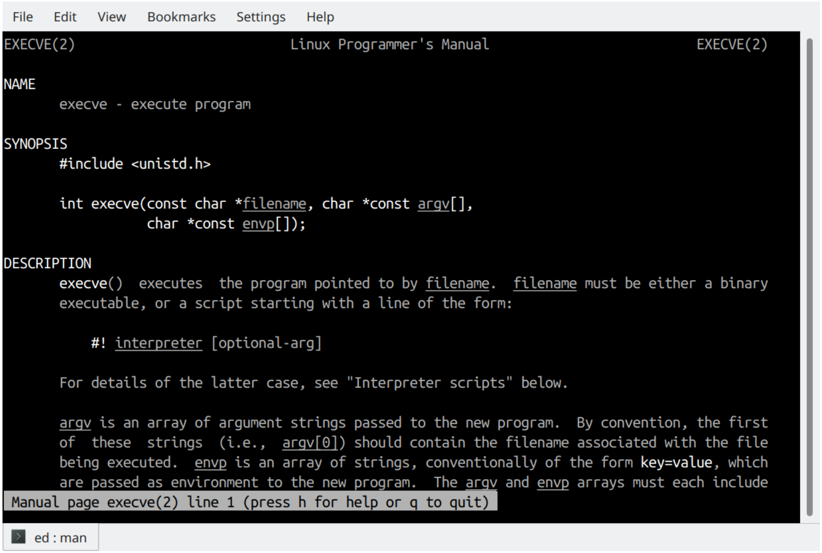
```
EXECVE(2)                  Linux Programmer's Manual                 EXECVE(2)

NAME
       execve - execute program

SYNOPSIS
       #include <unistd.h>

       int execve(const char *filename, char *const argv[],
                  char *const envp[]);

DESCRIPTION
       execve()  executes  the program pointed to by filename.  filename must be either a binary
       executable, or a script starting with a line of the form:

           #! interpreter [optional-arg]

       For details of the latter case, see "Interpreter scripts" below.

       argv is an array of argument strings passed to the new program.  By convention, the first
       of  these  strings  (i.e.,  argv[0]) should contain the filename associated with the file
       being executed.  envp is an array of strings, conventionally of the form key=value, which
       are passed as environment to the new program.  The argv and envp arrays must each include
 Manual page execve(2) line 1 (press h for help or q to quit)
```

Fork Man Page



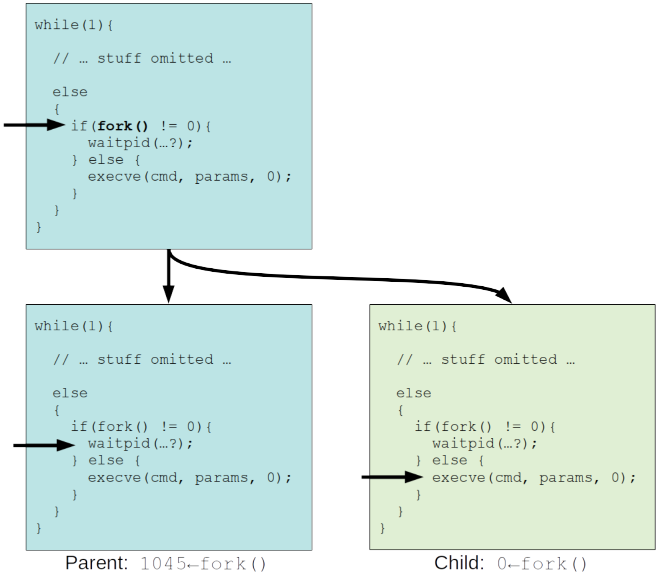
```
FORK(2)                    Linux Programmer's Manual                   FORK(2)

NAME
       fork - create a child process

SYNOPSIS
       #include <sys/types.h>
       #include <unistd.h>

       pid_t fork(void);

DESCRIPTION
       fork()  creates  a  new  process  by duplicating the calling process.  The new process is
       referred to as the child process.  The calling process  is  referred  to  as  the  parent
       process.

       The  child  process and the parent process run in separate memory spaces.  At the time of
       fork() both memory spaces have the same content.  Memory writes, file mappings (mmap(2)),
       and unmappings (munmap(2)) performed by one of the processes do not affect the other.

       The  child  process  is an exact duplicate of the parent process except for the following
       points:

 Manual page fork(2) line 1 (press h for help or q to quit)
```

Execve Man Page



```
EXECVE(2)                  Linux Programmer's Manual                 EXECVE(2)

NAME
       execve - execute program

SYNOPSIS
       #include <unistd.h>

       int execve(const char *filename, char *const argv[],
                  char *const envp[]);

DESCRIPTION
       execve()  executes  the program pointed to by filename.  filename must be either a binary
       executable, or a script starting with a line of the form:

           #! interpreter [optional-arg]

       For details of the latter case, see "Interpreter scripts" below.

       argv is an array of argument strings passed to the new program.  By convention, the first
       of  these  strings  (i.e.,  argv[0]) should contain the filename associated with the file
       being executed.  envp is an array of strings, conventionally of the form key=value, which
       are passed as environment to the new program.  The argv and envp arrays must each include
 Manual page execve(2) line 1 (press h for help or q to quit)
```

Fork() Diagram



```
while(1){

  // … stuff omitted …

  else
  {
    if(fork() != 0){
      waitpid(…?);
    } else {
      execve(cmd, params, 0);
    }
  }
}
```

```
while(1){

  // … stuff omitted …

  else
  {
    if(fork() != 0){
      waitpid(…?);
    } else {
      execve(cmd, params, 0);
    }
  }
}
```

```
while(1){

  // … stuff omitted …

  else
  {
    if(fork() != 0){
      waitpid(…?);
    } else {
      execve(cmd, params, 0);
    }
  }
}
```

Parent: `1045←fork()`

Child: `0←fork()`