CSE321 - Operating Systems Lab

Lab 04: System Calls, Process Creation

System Calls:

A system call is a request for the operating system to do something on behalf of the user's program.

- open()
- read()
- write()
- close()
- wait()
- fork()
- exec()

open():

open() lets you open a file for reading, writing, or reading and writing.

```
int open(file_name, mode)
```

where file_name is a pointer to the character string that names the file and mode defines the file's access permissions if the file is being created.

Different parameters for mode:

Value	Meaning
O_RDONLY	Open the file so that it is read only.
O_WRONLY	Open the file so that it is write only.
O_RDWR	Open the file so that it can be read from and written to.
O_APPEND	Append new information to the end of the file.
O_TRUNC	Initially clear all data from the file.
O_CREAT	If the file does not exist, create it. If the O_CREAT option is used, then you must include the third parameter.
O_EXCL	Combined with the O_CREAT option, it ensures that the caller *must* create the file. If the file already exists, the call will fail.

read() and write():

Both read() and write() take three arguments. Their prototypes are:

```
int file_descriptor;
char *buffer_pointer;
unsigned transfer_size;
int read(file_descriptor, buffer_pointer, transfer_size)
```

```
int file_descriptor;
char *buffer_pointer;
unsigned transfer_size;
int write(file_descriptor, buffer_pointer, transfer_size)
```

file_descriptor identifies the I/O channel, buffer_pointer points to the area in memory where the data is stored for a read() or where the data is taken for a write(), and transfer_size defines the maximum.

close():

Close a channel using the close() system call. The prototype for the close() system call is:

```
int close(file_descriptor)
```

lseek():

lseek() system call repositions the read/write file offset. Examples -

- `lseek(fd,5,SEEK_SET)` – this moves the pointer 5 positions ahead starting from the beginning of the file
- `lseek(fd,5,SEEK_CUR)` – this moves the pointer 5 positions ahead from the current position in the file
- `lseek(fd,-5,SEEK_CUR)` – this moves the pointer 5 positions back from the current position in the file
- `lseek(fd,-5,SEEK_END)` -> this moves the pointer 5 positions back from the end of the file

Implementation of open(), read(), write(), lseek() and close() functions:

```
#include<stdio.h>
#include<fcntl.h>
 int main()
 {
   int fd;
   char buffer[80];
   static char message[] = "Hello, world";
   fd = open("myfile",O_RDWR);
   if (fd != -1)
     {
       printf("myfile opened for read/write access\n");
       write(fd, message, sizeof(message));
       lseek(fd, 0, 0); /* go back to the beginning of the file */
       read(fd, buffer, sizeof(message)
       printf(" %s was written to myfile \n", buffer);
       close (fd);
   }
}
```

fork():

- When the fork system call is executed, a new process is created which consists of a copy of the address space of the parent.
- The return code for fork is zero for the child process and the process identifier of child is returned to the parent process.
- On success, both processes continue execution at the instruction after the fork call.
- On failure, -1 is returned to the parent process.

```
#include <sys/types.h>
main()
{
  pid_t pid;
  pid = fork();
  if (pid == 0)
    printf("\n I'm the child process");
  else if (pid > 0)
    printf("\n I'm the parent process. My child pid is %d", pid);
  else
    perror("error in fork");
}
```

wait()

The wait system call suspends the calling process until one of its immediate children terminates.

If the call is successful, the process ID of the terminating child is returned.

Zombie process—a process that has terminated but whose exit status has not yet been received by its parent process.

- the process will remain in the operating system's process table as a zombie process, indicating that it is not to be scheduled for further execution
- But that it cannot be completely removed (and its process ID cannot be reused)

pid_t wait(int *status)

Where status is an integer value where the UNIX system stores the value returned by child process

Implementing wait() system call:

```
#include <stdio.h>
void main()
{
  int pid, status;
  pid = fork();
  if(pid == -1)
 {
    printf("fork failed\n");
    exit(1);
 }
 if(pid == 0)
 { /* Child */
   printf("Child here!\n");
 }
 else
 { /* Parent */
   wait(&status);
   printf("Well done kid!\n");
 }
}
```

exec()

- Typically the exec system call is used after a fork system call by one of the two processes to replace the process' memory space with a new executable program.
- The new process image is constructed from an ordinary, executable file
- There can be no return from a successful exec because the calling process image is overlaid by the new process image

Implementation in C:

execl Takes the path name of an executable program (binary file) as its first argument. The rest of the arguments are a list of command line arguments to the new program (argv[]). The list is terminated with a null pointer:

```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
int main(){
  printf("1 \n");
  pid_t pid = fork();
  if(pid == 0) execl("bin/ls/","ls", NULL);
  else if(pid>0) execl("bin/pwd/","pwd", NULL);
}
```

Example 2: (Running different program from a specific program)

```
#include<stdio.h>
int main(int argc, char* argv[]){
  printf("Program-1 arguments passed: %d", argc);
  for(int i=0; i<argc; i++){
    printf("%s ", argv[i]);
  }
}
```

program1.c

$gcc program1.c -o program1

```
#include<stdio.h>
```

```
#include<unistd.h>
int main(){
  printf("Program-2 Running...");
  pid_t pid, status;
  pid = fork();
  if(pid == 0)
    execl("home/john/Desktop/","program1", 'a', 'b', 'c', 'd', NULL);
  else if(pid>0){
    wait(&status);
    execl("bin/pwd/","pwd", NULL);
  }
}
```

program2.c

$gcc program2.c -o program2

```
Program-2 Running...
Program-1 arguments passed:5
a
b
c
d
home/john/Desktop/
```

Lab Tasks:

Task - 1:

```
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

#define FORK_DEPTH 3

main()
{
  int i, r;
  pid_t my_pid;

  my_pid = getpid();

  for (i = 1; i <= FORK_DEPTH; i++) {

    r = fork();

    if (r > 0) {
      /* we're in the parent process after
         successfully forking a child */

      printf("Parent process %d forked child process %d\n",my_pid, r);

    } else if (r == 0) {

      /* We're in the child process, so update my_pid */
      my_pid = getpid();

      /* run /bin/echo if we are at maximum depth, otherwise continue loop */
      if (i == FORK_DEPTH) {
        r = execl("/bin/echo","/bin/echo","Hello World",NULL);

        /* we never expect to get here, just bail out */
        exit(1);
      }
    } else { /* r < 0 */
      /* Eek, not expecting to fail, just bail ungracefully */
      exit(1);
    }
  }
}
```

Answer the following questions about the code below.

1. What is the value of i in the parent and child after fork.
2. What is the value of my_pid in a parent after a child updates it?
3. What is the process id of /bin/echo?
4. Why is the code after execl not expected to be reached in the normal case?
5. How many times is *Hello World* printed when FORK_DEPTH is 3?
6. How many processes are created when running the code (including the first process)?

Task - 2:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char teststr[] = "The quick brown fox jumps over the lazy dog.\n";

main()
{
  int fd;
  int len;
  ssize_t r;


  fd = open("testfile", O_WRONLY | O_CREAT, 0600);
  if (fd < 0) {
    /* just ungracefully bail out */
    perror("File open failed");
    exit(1);
  }

  len = strlen(teststr);
  printf("Attempting to write %d bytes\n",len);

  r = write(fd, teststr, len);

  if (r < 0) {
    perror("File write failed");
    exit(1);
  }
  printf("Wrote %d bytes\n", (int) r);

  close(fd);
```

```
}
```

1. What does the following code do? What will be the output?
2. In addition to O_WRONLY, what are the other 2 ways one can open a file?
3. What open return in fd, what is it used for? Consider success and failure in your answer.

Task - 3:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char teststr[] = "The quick brown fox jumps over the lazy dog.\n";

main()
{
  int fd;
  int len;
  ssize_t r;
  off_t off;


  fd = open("testfile2", O_WRONLY | O_CREAT, 0600);
  if (fd < 0) {
    /* just ungracefully bail out */
    perror("File open failed");
    exit(1);
  }

  len = strlen(teststr);
  printf("Attempting to write %d bytes\n",len);

  r = write(fd, teststr, len);

  if (r < 0) {
    perror("File write failed");
    exit(1);
  }
  printf("Wrote %d bytes\n", (int) r);

  off = lseek(fd, 5, SEEK_SET);
  if (off < 0) {
    perror("File lseek failed");
    exit(1);
```

```
  }

  r = write(fd, teststr, len);

  if (r < 0) {
    perror("File write failed");
    exit(1);
  }
  printf("Wrote %d bytes\n", (int) r);

  close(fd);

}
```

The following code is a variation of the previous code that writes twice.

1. How big is the file (in bytes) after the two writes?
2. What is lseek() doing that is affecting the final file size?
3. What over options are there in addition to SEEK_SET?.

Task - 4:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

main()
{
  int r;
  r = chdir("..");
  if (r < 0) {
    perror("Eek!");
    exit(1);
  }
  r = execl("/bin/ls","/bin/ls",NULL);
  perror("Double eek!");
}
```

Compile and run the following code.

1. What do the following code do?
2. After the program runs, the current working directory of the shell is the same. Why?
3. In what directory does /bin/ls run in? Why?