OS Project - Part 1

Selma Imširović
Amir Bašović

Task 1.5:

Question 1.5.1: Do the following actions require the OS to use kernel mode or user mode is sufficient? Explain.

1. A program wants to read from disk.
2. Reading the current time from the hardware clock.

Answer:

1. When a program wants to read from disk, **it requires the OS to use kernel mode**. This is because for the user mode the access to peripheral/hardware devices (including disk) is denied. For this action the OS needs to use the kernel mode because only in this mode it has the full access to the hardware. When the program reads from disk, it makes a call to the OS to access the hardware and reads data. A user mode then switches to kernel mode with full access to the disk. This call is also known as a system call.

1. If a program wants to read the current time from the hardware clock, it also **requires the OS to use the kernel mode** because it is interacting with some hardware component (in our case it is the hardware clock), which is privileged and for the user mode it is a denied action.

Question 1.5.2: Explain the purpose of a system call. There are different sets of system calls: list them and give at least 2 examples of a system call for each category.

Answer:

A system call is a way in which programs that are in user mode interact with the OS. If a program wants to execute some action for which it is not privileged because of the user mode (interactions with hardware, access to memory locations etc.), it makes a call to the OS to access those resources by switching to kernel mode. These calls are system calls and they serve as interfaces between the program and the OS to allow user mode processes to access some privileged resources.

Different sets of system calls:

1. **Process control**

1. Example **fork()** → used to create a new child process from the parent process. When the fork() is called, the current process duplicates itself resulting in two processes running at the same time but with different process IDs :

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

int main() {
   pid = fork();

   if (pid < 0) {
      perror(stderr, "Fork failed!\n");
      exit(1);
   } else if (pid == 0) {
      // Child process
      printf("This is the child process with PID = %d\n", getpid());
   } else {
      // Parent process
      printf("This is the parent process with PID = %d\n", getpid());
   }
}
```

2. Example **exec()** → in comparison with fork() system call, exec() family of system calls replaces the current process with a whole different process, but the new process has the same process ID as the current process in which exec() is called.

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

int main()
{
      printf("Hello from the current process!\n");

//Assuming that process2 is another program that calculates the sum of arguments sent
      char *args[] = {"process2", "10", "30", NULL};
      execv("./process2", args);
}
```

2. File systems

1. Example **open()** → system call that opens a file. It comes with some flags eg. O_WRONLY (for writing and reading), O_CREAT (for creating if the file doesn't exist) and permission digit numbers.

2. Example **write()** → system call for writing into files

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

int main() {

 int fd;
 int wb;

fd=open("osproject.txt", O_WRONLY | O_CREAT, 0666);
char message[]="First OS project by Amir and Selma.";
wb=write(fd, message, sizeof(message)-1);

        if (wb<0){
            perror("Writing failed");
            exit(1);
        } else {
            printf("Writing successful");
            close(fd);
        }
}
```

3. Memory management system calls

1. Example **brk()**
2. Example **sbrk()**

→ both system calls are used to change the amount of allocated memory space of the current process.

4. Interprocess communication system calls

1. Example **pipe()** → used for communication between processes when one process needs to send data to another eg. when output of one process needs to serve as an input of the second process,

then the pipe() system comes into play, to pipe the output of one process to another.

2. Example **socket()** → creates communication endpoints that can be used for communication between processes, but over a network.

5. Device management system calls

1. Example **readConsole()** → reads character input from a console's temporary storage.
2. Example **writeConsole(**) → writes character data to a console's screen.
   → both provide a mechanism for storing and managing user input from and to the console.

Question 1.5.3: What are the possible outputs from the printed program?

Answer:

Possible outputs from the printed program are:

1. fork() fails, exec() fails:

   Output: "Hello4"

2. fork() fails, exec() succeeds:

   Output: not possible for exec() to succeed if fork() fails, because exec() is only called if for() succeeds.

3. fork() succeeds, exec() succeeds:

   Output: "Hello1"
   "some executable"
   "Hello3"

4. fork() succeeds, exec() fails:

   Output: "Hello1"
   "Hello2"

“Hello3”

5. Parent process execution (ret>0):

   Output: “Hello3”

File list:

task1-1Advanced.c
task1-1Basic.c
task1-2Advanced.c
task1-2Basic.c
task1-2Intermediate.c
task1.3a.c
task1.3b.c
intermediate1.3.c
intermediateTextFile.txt
additionally1.3.c
task1-4.c

Questions:

An outline of what has been done in the assignment:

1. Designed and implemented a basic shell interface capable of executing other programs and supporting built-in functions.

2. Implemented basic functionality for shell programs like cp, history, free, and fortune, with additional options and arguments for some.

3. Allowed piping or redirecting output to a text file for advanced functionality.

4. Implemented system calls such as fork(), wait(), exec(), and execlp() within a C-programming example.

5. Explored and implemented a forkbomb within the C-programming example. Enhanced the shell with color and gave it a distinct name.

6. Provided concise and descriptive answers to questions about OS actions requiring kernel mode or user mode, the purpose of system calls, and possible outputs from a given program scenario.

Compiling instructions:

Task 1.1

1. Navigate to folder using **cd** command.
2. Compile command **gcc -o task1-1 task1-1.c**
3. Executing program command **./task1-1**

Task 1.2

1. Navigate to folder using **cd** command.
2. Compile command **gcc -o task1-2 task1-2.c**
3. Copy command **cp source_file destination_file**
4. History command **history.**
5. Free command **free.**
6. Fortune command **fortune.**
7. Piping : **command1 | command2**
8. Redirect command to text file **command > output.txt**
9. This would redirect output of “command” to text file **command >> output.txt**

Task 1.3

1. Navigate to the folder using **cd** command.
2. Compile the first program from basic part with command **gcc -Wall -o process1 task1.3a.c**
3. Compile the second program from basic part with command **gcc -Wall -o process2 task1.3b.c**
4. Run the first program with command **./process1**
5. Compile the program from intermediate part with command **gcc -Wall -o intermediate intermediate1.3.c**
6. Run the program with command **./intermediate**
7. Compile the program from additionally part with command **gcc -Wall -o forkbomb additionally1.3.c**
8. Run the program with command **./forkbomb**

Task 1.4

1. Navigate to folder using **cd** command.
2. Compile command **gcc -o task1-4 task1-4.c**
3. Executing command **./task1-4**

List of challenges:

1. User input parsing.
2. Command execution.
3. Error handling.
4. Memory management.
5. Concurrency and pipelining.
6. Performance optimization.
7. The rfork() system call → **since it is not supported by Unix-like systems we had to implement the fork() system call instead**.
8. The execv() system call and its parameters.

Additional sources:

- https://medium.com/@winfredngina kilonzo/guide-to-code-a-simple-shell-in-c-bd4a3a4c41cd
- https://www.youtube.com/watch?v=4jYFqFsu03A
- https://minnie.tuhs.org/CompArch/Lectures/week05.html#:~:text=At%20the%20same%20time%2C%20a,can%20is%20the%20operating%20system.
- https://www2.cs.uregina.ca/~hamilton/courses/330/notes/unix/memory/brk-notes.txt
- https://www.geeksforgeeks.org/different-types-of-system-calls-in-os/
- Operating Systems: Three Easy Pieces (Remzi Arpaci-Dusseau and Andrew Arpaci-Dusseau) - Book literature