CL2006 - Operating Systems Spring 2024
LAB # 4 MANUAL (Common)

Please note that all labs' topics including pre-lab, in-lab and post-lab exercises are part of the theory and labsyllabus. These topics will be part of your Midterms and Final Exams of lab and theory.

Objectives:

Demonstrate System Calls Programmatically:

1. Develop an understanding of system calls by implementing various functions programmatically and explore the functionality and usage of each system call through practical coding exercises, emphasizing their role in process management, information retrieval, and file manipulation.

2. Implement system calls such as fork (), wait(), exit(), exec(), sleep(), getpid(), getppid(), alarm(), open(), read(), write(), and close() to perform essential operating system tasks.

Lab Tasks:

Use of system calls related to the following:
1. Process creation and control.
2. Information retrieval
3. File creation and manipulation.

Delivery of Lab contents:

Strictly following the following content delivery strategy. Ask students to take notes during the lab.

1st Hour
- Pre-Lab (up to 15 minutes)
- System Call introduction (15 minutes)
- C program structure for making system calls (30 minutes)

2nd Hour
- Process management (60 minutes)

3rd Hour
- Information retrieval, and File manipulation (60 minutes)

Initial Document: Hamza Yousuf (January 2019). This version by: Nadeem Kafi (09/02/2024)
DEPARTMENT OF COMPUTER SCEICEN, FAST-NU, KARACHI

** ChatGPT is heavily used to make the contents of this document along with other Internet sources.

EXPERIMENT 4

System Calls

What are system calls?

A system call is a mechanism that allows a user-level program to request a service or operation from the kernel (the core part of the operating system). These services can range from low-level tasks like accessing hardware devices to higher-level operations like creating or terminating processes.

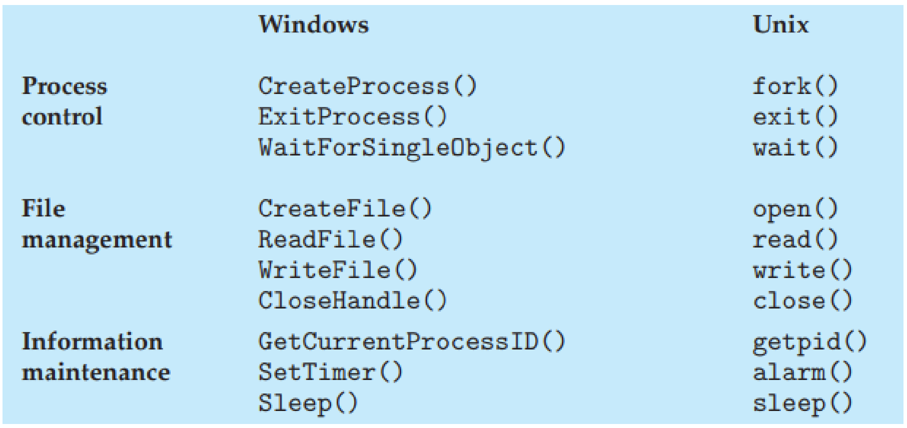
1. **Communication with the Kernel**: The kernel is the central component of an operating system responsible for managing hardware resources and providing essential services to user programs. User programs, including applications and utilities, operate in a restricted environment known as user mode. Direct access to hardware resources is prohibited in user mode for security and stability reasons.

2. **Transition to Kernel Mode:** When a user program requires a service that only the operating system kernel can provide, it needs to transition from **user mode** to a more **privileged mode** called kernel mode. System calls provide a controlled mechanism for this transition. When a system call is invoked, the program **switches from user mode to kernel mode** to execute the requested operation.

3. **Requesting Operating System Services:** System calls act as an interface between user programs and the operating system kernel. They allow user programs to request various services or operations provided by the kernel. Examples of common system calls include:
   - Opening, reading from, writing to, and closing files.
   - Creating, executing, and terminating processes.
   - Allocating and managing memory.
   - Interacting with hardware devices such as printers and network interfaces.

4. **Implementation by the Kernel**: Each system call has a unique identifier or number associated with it, known as a syscall number. This number is used by the kernel to identify the requested service. The kernel contains implementations for handling different system calls. When a system call is invoked, the kernel executes the corresponding routine to perform the requested operation.

5. **Return Values and Error Handling**: System calls typically return a value to the calling program, indicating the success or failure of the requested operation. If an error occurs during the execution of a system call, the kernel returns a special error code, allowing the user program to handle the error gracefully.



	Windows	Unix
Process control	`CreateProcess()`	`fork()`
	`ExitProcess()`	`exit()`
	`WaitForSingleObject()`	`wait()`
File management	`CreateFile()`	`open()`
	`ReadFile()`	`read()`
	`WriteFile()`	`write()`
	`CloseHandle()`	`close()`
Information maintenance	`GetCurrentProcessID()`	`getpid()`
	`SetTimer()`	`alarm()`
	`Sleep()`	`sleep()`

Figure 1 System calls for Windows and Unix/Linux Operating Systems

C program structure for doing System Calls

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

int main () {
    // Declare variables and necessary data structures

    // Make explicit system calls

    // Handle errors if any

    // Perform necessary operations using system calls

    return 0;
}
```

This basic structure provides a framework for writing C programs that utilize explicit Linux/UNIX system calls, enabling developers to interact directly with the underlying operating system for performing various tasks. Explanation of various part of the programs follows:

- **#include <stdio.h>:** The standard input-output library for basic input and output operations.
- **#include <stdlib.h>:** The standard library for general utilities and memory allocation.
- **#include <unistd.h>:** The POSIX (Portable Operating System Interface) standard header file for various system calls, such as process management (fork(), exec(), exit(), etc.).
- **#include <sys/types.h>:** Definitions for various data types used in system calls, such as pid_t for process IDs.
- **#include <sys/wait.h>:** Definitions for functions related to process management and waiting for child processes to terminate.
- **#include <errno.h>:** Definitions for error numbers that may occur during system calls.
- Inside the main() function:
  - Declare necessary variables and data structures for holding information related to system calls and their results.
  - Make explicit system calls as required for the intended functionality of the program. These system calls could involve process management, file operations, memory allocation, etc.
  - Handle errors that may occur during system calls using error-checking mechanisms like errno and appropriate error-handling techniques.
  - Perform necessary operations using the results obtained from system calls.
  - return 0. Indicates successful termination of the program by returning a status code of 0 to the operating system.

Process Creation:

The processes in most systems can execute concurrently, and they may be created and deleted dynamically. Thus, these systems must provide a mechanism for process creation and termination.

1. **fork()**
   - Has a return value.
   - Parent process => invokes fork () system call
   - Continue execution from the next line after fork ()
   - Has its own copy of any data.
   - Return value is > 0   //it's the process id of the child process. This value is different. from the Parents own process id.
   - Child process => process created by fork() system call
   - Duplicate/Copy of the parent process   //LINUX
   - Separate address space
   - Same code segments as parent process
   - Execute independently of parent process
   - Continue execution from the next line right after fork()
   - Has its own copy of any data
   - Return value is 0

2. **wait ()**
   - Used by the parent process
   - Parent's execution is suspended
   - Child remains its execution
   - On termination of child, returns an exit status to the OS
   - Exit status is then returned to the waiting parent process   //retrieved by wait ()
   - Parent process resumes execution
   - #include <sys/wait.h>
   - #include <sys/types.h>

3. **exec()**
   - The exec() system call replaces the current process image with a new process image, which means it loads a new program into memory and starts its execution. Called by an already existing process.
   - It has several variants, such as execl(), execv(), execle(), execve(), etc., each with different ways of passing arguments to the new program.
   - Here are the general steps involved in using the exec() system call:
     - Open the Target Program File: The first step is to locate and open the file of the program you want to execute. This could be a binary executable file, a script, or any other executable file.
     - Load the Program into Memory: The exec() system call loads the program into the current process's memory space, replacing the current program image.
     - Set Up Execution Context: The system sets up the execution context for the new program, including initializing the program counter, setting up the stack, and preparing other resources needed for execution.
     - Start Execution: Finally, the new program starts its execution from its entry point, as defined in its binary executable file.
   - Has an exist status but cannot return anything (if exec() is successful) to the program that made the call i.e. parent process
   - Return value is -1 if not successful.
   - Overlay => replacement of a block of stored instructions or data with another. int execlp(char const *file_path, char const *arg0,);
   - Arguments beginning at arg0 are pointers to arguments to be passed to the new process.

- The first argument arg0 should be the name of the executable file.
- Example: execlp(/bin/ls , ls ,NULL) //lists contents of the directoryHeader file used -> unistd.h

4. **exit():**

- The exit() system call is used to terminate a process immediately from any point within the program, not just within the main() function. EXIT_SUCCESS // integer value = 0, EXIT_FAILURE // integer value = 1
- It takes an integer argument known as the exit status, which is returned to the parent process or environment to indicate the termination status of the program.
- The exit() system call performs cleanup tasks, such as closing open files and releasing allocated memory, before terminating the process.

- Calling exit() directly causes the program to exit immediately from the point where it is called, bypassing any further execution of the program.
- Process related:
  - It returns exit status, which is retrieved by the parent process using wait () command.
  - OS reclaims resources allocated by the terminated process (dead process) Typically performs clean-up operations within the process space before returning control.
  - Terminates the current process without any extra program clean-up.
  - Usually used by the child process to prevent from erroneously release of resources belonging to the parent process.

Information Maintenance:

i. **sleep()**

- Process goes into an inactive state for a time period
- Resume execution if
- Time interval has expired
- Signal/Interrupt is received
- Takes a time value as parameter (in seconds on Unix-like OS and inmilliseconds on Windows OS)
- sleep(2) // sleep for 2 seconds in Unix
- Sleep(2*1000) // sleep for 2 seconds in Windows

ii. **getpid() // returns the PID of the current process**

- getppid() // returns the PID of the parent of the current process
- Header files to use
- #include <sys/types.h>
- #include <unistd.h>
- getppid() returns 0 if the current process has no parent

iii. **alarm()**

- Use to set an alarm timer for a process. It takes an argument representing the number of seconds after which the alarm signal will be delivered to the calling process.
- The signal, known as SIGALRM, interrupts the normal execution of the process and can be used to perform specific actions or handle timeouts. The program to respond accordingly, such as terminating the operation or executing a signal handler function.
- Common use cases for alarm() include implementing timeouts for certain operations, scheduling periodic tasks, or triggering events after a specified delay.
- Additionally, the return value of alarm() provides information about any previously set alarm timer, allowing programs to query or cancel existing timers as needed. Overall, alarm() is a

versatile system call that facilitates time-based programming and event handling in Unix-based systems.

File Management:

1. **open ()**

- **Specify File Path and Access Mode:** Provide the file path of the file you want to open as well as the access mode (e.g., read-only, write-only, read-write).
- **Check File Permissions:** The kernel checks whether the process has the necessary permissions to open the file based on the access mode specified.
- **Locate File Descriptor:** If permissions are granted, the kernel locates an available file descriptor (an integer representing the file) for the file within the process's file descriptor table.
- **Open File and Return Descriptor:** The kernel opens the file and associates it with the file descriptor. If successful, it returns the file descriptor to the calling process.

2. **read ()**

- **Specify File Path and Access Mode:** Provide the file path of the file you want to open as well as the access mode (e.g., read-only, write-only, read-write).
- **Check File Permissions:** The kernel checks whether the process has the necessary permissions to open the file based on the access mode specified.
- **Locate File Descriptor:** If permissions are granted, the kernel locates an available file descriptor (an integer representing the file) for the file within the process's file descriptor table.
- **Open File and Return Descriptor:** The kernel opens the file and associates it with the file descriptor. If successful, it returns the file descriptor to the calling process.

3. **write ()**

- **Specify File Descriptor and Data:** Provide the file descriptor of the file to which you want to write data and the data to be written.
- **Check File Descriptor Validity:** The kernel verifies that the file descriptor provided is valid and points to an open file.
- **Write Data:** The kernel writes the specified data to the file associated with the file descriptor.
- **Update File Position:** After writing, the kernel updates the file position indicator associated with the file descriptor to reflect the new position in the file.
- **Return Number of Bytes Written:** The write() system call returns the number of bytes successfully written to the file to the calling process.

4. **close ()**

- **Specify File Descriptor:** Provide the file descriptor of the file you want to close.
- **Check File Descriptor Validity:** The kernel verifies that the file descriptor provided is valid and points to an open file.
- **Release Resources:** The kernel releases any resources associated with the file descriptor, such as closing the file and freeing memory.
- **Invalidate File Descriptor:** After closing the file, the kernel invalidates the file descriptor, making it available for reuse.
- **Return Status:** The close () system call returns a status indicating whether the file was successfully closed or if an error occurred to the calling process.

In-lab problems:

1. Write a program that creates two child processes. One child process should print its process ID (PID), and the other child process should print its parent process ID (PPID). The parent process should wait for both child processes to terminate before ending. **Hint: fork (), wait ()**

2. Create a program that creates a child process. The child process prints “I am a child process” 100 times in a loop. Whereas the parent process prints “I am a parent process” 100 times in a loop.

3. Develop a program that prints the current process ID (PID) and the parent process ID (PPID) of the running process. Additionally, implement a function to retrieve and print the user ID (UID) of the current user executing the program. **Hint: getpid (), getppid ()**

4. Create a program named stat that takes an integer array as command line argument (deliminated by some character such as $). The program then creates 3 child processes each of which does exactly one task from the following: i) Adds them and print the result on the screen. (done by child 1), ii) Shows the average on the screen. (done by child 2), iii) Prints the maximum number on the screen. (done by child 3)

5. Create a program that reads input from a text file named "input.txt" and writes the content to another text file named "output.txt". Ensure proper error handling for file opening, reading, and writing operations. **Hint: open (), read (), write (), close ()**

6. Invoke at least 4 commands from your programs, such as Cp, mkdir, rmdir, etc (The calling program must not be destroyed)

7. Write a program that demonstrates the usage of the fork() system call to create a child process. The child process should execute a shell command to list all files in the current directory, while the parent process waits for the child to terminate. Ensure proper synchronization between parent and child processes. **Hint: fork (), execlp ()**

8. Implement a program that utilizes the sleep()` and `alarm()` system calls. The program should initially set an alarm for 5 seconds and then enter a sleep loop. Within this loop, the program should print a message every second. When the alarm signal is received after 5 seconds, the program should terminate gracefully. **Hint: sleep (), alarm ()**

Post-Lab Questions:

1. Write a program which uses fork () system-call to create a child process. The child process prints the contents of the current directory, and the parent process waits for the childprocess to terminate.

2. Write a program which prints its PID and uses fork () system call to create a child process. After fork () system call, both parent and child processes print what kind of processthey are and their PID. Also, the parent process prints its child's PID, and the child process prints its parent's PID.

3. Develop a program that copies the contents of one file into another file using system calls. The program should accept two file paths as command-line arguments: the source file to be copied from and the destination file to be copied to. Ensure proper error handling for file opening, reading, and writing operations.

4. Write a program that lists all files and directories in the current directory using system calls. The program should traverse the directory structure recursively and print the names of all files and directories found, along with their respective types (file or directory).

Solution hints for problem # 4:

1. Useful System Calls:
   - opendir(): Use this system call to open a directory stream for the current directory.
   - readdir(): Call this system call to read the next entry from the directory stream opened by opendir().
   - closedir(): Use this system call to close the directory stream after reading all entries.

2. Looping through Directory Entries:
   - Use a loop to iterate through each directory entry returned by readdir().
   - Check the type of each entry using the st_mode field of the struct dirent structure returned by readdir().

3. Handling Subdirectories Recursively:
   - For each directory entry that is a directory itself, consider recursively calling the same function to list its contents.
   - Ensure proper termination conditions to avoid infinite recursion.

4. Printing File and Directory Names:
   - Print the names of files and directories using the d_name field of the struct dirent structure returned by readdir().
   - Determine whether an entry is a file or a directory by checking its type using st_mode.