East West University
Department of Computer Science and Engineering
Lab Sheet of CSE325

Lab Task : System Call

PART: B

OBJECTIVE:

- To understand about Linux system calls and their use in processes

BACKGROUND:

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

fork()

- 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

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

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
  Parent process resumes execution
  #include <sys/wait.h>
  #include <sys/types.h>

exit()

- Process terminates its execution by calling the exit() system call
- It returns exit status, which is retrieved by the parent process using wait() command
  EXIT_SUCCESS // integer value = 0
  EXIT_FAILURE // integer value = 1
- 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

Problem 01:

```
#include <stdio.h>
#include <sys/types.h>
int main()
{
pid_t pid;
pid=fork();
if(pid==0){
printf("I'm in a child process \n\n");
}
else if (pid>0){
printf("I'm in the parent process \n\n");
}
else
{
printf("Error \n\n");
}
return 0;
}
```

Explanation:

fork() creates a new process. If fork() returns a negative value, the creation of the child process was unsuccessful. If fork() returns zero, it means the code is running in the child process. If fork() returns a positive value, it means the code is running in the parent process, and the value is the PID of the child process.

Problem 02:

```
#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>
```

```
int main() {
    fork();
    fork();
    fork();
    printf("Hello from process PID: %d\n", getpid());
    return 0;
}
```

Explanation:

Each fork() call creates a new process. The number of processes created is (2^n), where (n) is the number of fork() calls. In this example, 3 fork() calls create (2^3 = 8) processes.

Problem 03: Modify the basic fork example to make the parent process wait for the child process to complete using the wait() system call.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
int main() {
    pid_t pid = fork();
    if (pid < 0) {
        // Fork failed
        perror("Fork failed");
        return 1;
    } else if (pid == 0) {
        // Child process
        printf("Hello from the child process! PID: %d\n", getpid());
    } else {
        // Parent process
        wait(NULL); // Wait for the child process to complete
        printf("Hello from the parent process! PID: %d\n", getpid());
    }
    return 0;
}
```

Try 1: Write a program that creates a child process and makes the child process print numbers from 1 to 10.

Try 2: Write a program that creates two child processes. Each child process should print its own PID and the PID of its parent.

Try 3: Write a program that creates a chain of processes where each child creates another child. The chain should be of length 3, and each process should print its PID and the PID of its parent.

Try 4: 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 process they are and their PID. Also the parent process prints its child's PID and the child process prints its parent's PID.