Here you are given detailed explanations of the system calls & processes-related topics. Your assignment is given after these explanations as a separate document and requires that you read and understand all of these concepts.

The current process can create a child process by using the system call `fork`. This primitive causes the current process to be duplicated. It is of the following form:

```c
#include <unistd.h>
#include <sys/types.h>

pid_t fork (void);

Returns: 0 in child, PID of child in parent, -1 on error
```

When `fork` is called, the current process is duplicated: a copy which conforms to the original, except its identifier, and that of its parent, is created. When `fork` is returned, two processes, the parent and child, execute the same code.

The primitive `fork` returns the value 0 to the child process, and it returns the identifier (PID) of the created process to the parent process. It is therefore vital to test the returned value in order to distinguish the code which should be executed by the parent process from that which should be executed by child process. In the case of failure, `fork` returns a value -1. Here’s an example program uses the system call `fork` to create a child process.

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

void main(void)
{
    pid_t pid;
    pid = fork();
    if (pid == -1)
        perror ("fork");
    else if ( pid == 0)
        printf("I am the child : pid = %d\n", pid);
    else
        printf("I am the parent : pid = %d\n", pid);
}
```

In this example program, the first `printf` is executed in the child process and the second `printf` is executed in the parent process.
The current process terminates automatically when it finishes executing the function main, by using the instruction `return`. There is also a system call to explicitly stop execution:

```
#include <unistd.h>
void exit (int status);
```

The parameter status specifies a return code in the range 0 to 255, to communicate with the parent process. By convention, a process should return a value of 0 in the case of a normal termination, and a non-zero value in the case of termination following an error.

A process may wait for the termination of a child process by using the primitives `wait` and `waitpid`.

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait (int *status);
pid_t waitpid (pid_t pid, int *status, int options);
```

Both return: PID if OK, -1 on error

The primitive `wait` suspends execution of the current process until a child process has terminated. If a child process has already terminated, `wait` returns the result immediately. The primitive `waitpid` suspends execution of the current process until the child process specified by the parameter `pid` terminates. If a child process corresponding to `pid` has already terminated, `waitpid` returns the result immediately. The exit status of the child process is returned in the MSB byte of the int variable pointed by `*status`. If this information is not important, one may pass a null pointer as status.

Several system calls enable a process to retrieve its characteristic attributes. Some of them may be summarized as:

```
#include <unistd.h>
pid_t getpid (void);
pid_t getppid (void);
uid_t getuid (void);
```

Here `getpid()` returns a process its own process ID. `getppid()` returns a process its parent’s process ID. `getuid()` returns a process its user ID.

A new process, created by calling the primitive `fork`, is a copy based on its parent process, and therefore runs the same program. A system call allows a process to run a new program:

```
#include <unistd.h>
int execve (const char *pathname, const char *argv [], const char *envp[]);
```

The primitive `execve` triggers the execution of a new program. `pathname` specifies the name of the file to be executed, which must be a binary program or a file of commands beginning with the line `#!/interpreter_name`. The parameter `argv` should point to a string of characters representing an argument. The first element of the table should contain the name of the program, the elements following this should contain the arguments, and the final element should contain the value NULL. The parameter `envp` specifies the variables in the program environment. Each of the elements should contain the address of a character string of the form `name_of_variable=value`, and the final element of the table should contain the value NULL. The system call `execve` triggers the overwriting of segments of code, data and stack with those of the specified program. If the process is successful, there is therefore no return, since the calling process
then runs a new program. If a problem occurs, \texttt{execve} returns a value -1, and the variable \texttt{errno} can take appropriate values. Several library functions (not system calls) offering alternatives to \texttt{execve}, are also provided by Linux:

```c
#include <unistd.h>

int execl (const char *pathname , const char *arg , …/* (char *) 0 */ );
int execv (const char *pathname , char *const argv[ ] );
int execle (const char *pathname , const char *arg, … /* (char *) 0 char *const envp[ ]*/ );
int execvp (const char *filename , char *const argv[ ] );
```

For all these functions, the parameter \textbf{pathname} specifies the program to be run. The parameter \textbf{argv} of \texttt{execv} and \texttt{execvp} indicate the parameters to pass to the program, in the same way as for \texttt{execve}. The parameter \textbf{envp} indicates the environment variables to be passed to the program. Finally, for the functions \texttt{execel}, \texttt{execl}, and \texttt{execlp}, the program arguments are stated by explicit parameters: \textbf{arg} should contain the address of a character string representing the name of the program and should be followed by a list of pointers containing the argument addresses, and the last parameter on the list should be the value NULL.

The two functions \texttt{execlp} and \texttt{execvp} treat \texttt{pathname} as a simple command, in contrast to other functions and to \texttt{execve}: the search route associated with the current process (environment variable PATH) is used to search for the executable program specified.

```c
#include <sys/types.h>
#include <sys/wait.h>

char *env_init() = { "USER=unknown", "PATH=/tmp", NULL} ;

int main (void)
{
    pid_t pid;
    if ((pid = fork()) <0 )
        err_sys("fork error");
    else if ( pid == 0 ) { /* Specify pathname, specify environment */
        if (execle("/home/guest/bin/echoall", "echoall", "myarg1", "myarg2", (char*) 0, env_init ) <0 )
            err_sys("execle error");
    }
    if (waitpid(pid, NULL, 0) <0 )
        err_sys("wait error");
    if ( (pid = fork() )<0 )
        err_sys("fork error");
    else if (pid == 0) { /* Specify filename, inherit environment */
        if (execvp ("echoall", "echoall", "only 1 arg", (char *) 0) < 0)
            err_sys("execvp error");
    }
    exit(0);
}
```

An example demonstrating the \texttt{exec} functions are given above. In this sample program, we first call \texttt{execl}, which requires a pathname and a specific environment. The next call is to \texttt{execlp}, which uses a filename and passes the caller’s environment to the new program. The only reason the call to \texttt{execlp} works is because the directory /home/stevens/bin is one of the current path prefixes. Note also that we set the first bit argument, \texttt{argv[0]} is the new program, to be the filename component of the \texttt{pathname}. Some shells set this argument to be complete \texttt{pathname}. The name \texttt{echoall} is used just as a name of a binary executable file.

Mesut Ali Ergin

ergin@ics.yeditepe.edu.tr