CSCI 3753: Operating Systems

Linux System Programming

Process Creation: A new process can be created from inside a C/C++ program by using the `fork`() system call. The process that calls `fork`() is called the *parent process*, and the newly created process is called the *child process*. After the call, the parent process and the child process run concurrently.

```
#include <sys/types.h>
#include <unistd.h>
pid_t pid;
pid = fork();
```

The child process is an exact copy of the parent process, except for the following:

1. The child process has a unique process id (PID).
2. The child process has different process id of its creator (PPID).

The `fork` is called by the parent, but returns in both the parent and the child. In parent, it returns the process id of the child process, whereas in the child, it returns 0. No child process is created and -1 is returned, if `fork` fails.

```
pid_t pid;
if ((pid = fork()) == -1) exit(1); /* FORK FAILED */
if (pid == 0) {
      /*put code for child process here*/
      cout << "My process id = " << getpid() << endl;
      exit(0);
}
/*put code for parent process here*/
cout << "My process id = " << getpid() << endl;
```

Program execution: The child process is not restricted to executing a subset of the statements in the parent process. It can also execute another program by overlaying itself with an executable file. The target executable file is read in on top of the address space of the very process that is executing, overwriting it in memory, and execution continues at the entry point defined in the file. The result is that the child process begins to execute a new program, under the same execution environment as the old program, which is now replaced.

The program overlay can be done by any of the several versions of `exec` system calls, including `execl`, `execv`, and `execve`. The various `exec` routines differ in the type and number of arguments they take. Read man page of `exec` for details.

```
pid_t pid;
if ((pid = fork()) == -1) exit(1); /* FORK FAILED */
if (pid == 0) {
      /* Child process will execute the executable code in file a.out */
      execl("a.out", NULL);
      exit(0);
}
/*put code for parent process here*/
```

Synchronization of parent and child processes: After creating a child process, the parent process may run independently or elect to wait for the child process to terminate, by using the `wait` system call, before proceeding further.

```
#include <sys/types.h>
#include <sys/wait.h>
int pid = wait(union wait *status)
```

The `wait` system call searches for a terminated child of the calling process.

1. If there are no child processes, `wait` returns immediately with value -1.
2. If one or more child processes have terminated already, `wait` selects an arbitrary terminated child, stores its exit status in the variable *status*, and returns its process id.
3. Otherwise, `wait` blocks until one of the child processes terminates and then goes to 2.

Process termination: Every running process eventually comes to an end.

1. The process runs to completion and the function `main` returns.
2. The process calls the library routine `exit`, or the system call `_exit`.
3. The process encounters an execution error or receives an interrupt signal, causing its premature termination.

The argument to `_exit`/`exit` is part of the termination status of the process. Conventionally, a zero argument indicates normal termination and a non-zero argument indicates abnormal termination.

Interprocess communication: A *pipe* is a direct (in memory) I/O channel between two processes. Often it is used together with the system calls `fork`, `exec`, `wait`, and `exit` to make multiple processes cooperate and perform parts of a larger task.

```
int fildes[2];
pipe(fildes);
```

`read` and `write` system calls are used for I/O through pipes. *fildes*[0] is used for reading and *fildes*[1] is used for writing.

```
int fildes[2], pid;
if ((pipe(fildes)) == -1) exit(1);
if ((pid = fork()) == -1) exit(1);
if (pid == 0) { /* child process */
      char buf1[40];
      close(fildes[1]);
            ...
      read(fildes[0], buf1, 20);
            ...
      exit(0);
}
/* parent process */
char buf2[40];
close(fildes[0]);
      ...
write(fildes[1], buf2, 20);
      ...
wait(union wait *status);
```

Process suspension: Execution of a process may be suspended for some interval of time using the `sleep` command.

```
#include <unistd.h>
unsigned seconds;
unsigned r = sleep(seconds);
```

Some useful shell commands

`ps`: The `ps` command displays information about the existing processes in the system. This command has several command-line arguments. Read the man page for details.

`kill` *pid*: This command terminates an existing process whose process id is *pid*.