OS3-FORK-EXECVE-SIGNALS: 61 – 90

os3-fork-execve

61 [T / F] In Unix/Linux systems, a process can only be spawned by a parent process. Thus, when your computer boots up, actually there is the first process created by the OS, e.g. a shell process in recovery mode or the GUI process in normal mode.

62 [T / F] A shell is a actually a process. Inside a shell, you can run another shell as another process.

63 [T / F] One of the reasons why Unix/Linux systems require you to use two system calls fork() and execve() (instead a one simple system call, such as createNewProcess()) is to allow the shell program to manipulate the file descriptors of the child process before the child process runs the new program – (See os4-files for more).

64 [T / F] Every shell has a current working directory (CWD) context. Programs opened from within a shell will inherit the shell's CWD.

65 [T / F] After execve() completes successfully, the following line of code after the execve() call will be executed. (for example: execve(aNewProgram); i++; ...; here the i++ will be executed after aNewProgram completes.

66 [T / F] The "int rv=fork()" returns a zero to the parent process if the call is successful.

67 [T / F] For the parent process, the return value (rv) from calling fork() contains a random positive number.

68 [T / F] fork() always returns a non-negative integer value.

69 [T / F] getpid() is a syscall to get the process ID.

70 [T / F] Because the child process receives a return value of 0, the child does not have a way to know who the parent process is.

71 [T / F] If a program calls fork() N times consecutively, there will be $2^{N+1} - 1$ processes created including the original process.

72 [T / F] When no synchronization functions are called (e.g. no wait() or waitpid()), the parent and child processes run concurrently.

73 [T / F] Concurrency (non-deterministic execution) implies that you cannot guarantee the same output everytime.

74 [T / F] wait() waits for any of the childrens, and waitpid() waits for a specific child.

os3-signals:

75 [T / F] When you type something on your keyboard, the USB/bluetooth connection is sending a keyboard interrupt to the CPU and wakes up the OS to read the keystrokes.

76 [T / F] Signals can be sent by the OS to a process or from a process to another process.

77 [T / F] Processes by default cannot communicate, unless with inter-process communication (IPC) such

as signals, files, shared memory, etc.

78 [T / F] A process can send a signal to another process without the help of the OS.

79 [T / F] The concept of signal handler is similar to the concept of exception handler. Signal handlers are written in the OS code and exception handlers are written in the application/user code.

80 [T / F] printf("%8x", main) will print out the function address (logical address) of my main() function, i.e. the location of where my main() function is (virtually) located in the memory.

81 [T / F] signal(SIGsomething, my_sigsomething_handler) essentially is making a call to my_sigsomething_handler function. .

82 [T / F] A process' signal handler addresses are recorded in the interrupt table.

83 [T / F] A process' signal handler addresses are recorded in the process control block (PCB) of the process.

84 [T / F] When a process A makes a kill() system call to another process B, it means process A is shutting down process B.

85 [T / F] When a process A makes a kill() system call to another process B, the OS will return to A *while* B is handling the signal.

86 [T / F] If process A sends a signal X to process B, but B does not have X_sig_handler(), process A will die.

87 [T / F] Sending a signal creates a new *thread* (to handle the signal) in the destination process.

88 [T / F] The destination process' main thread has to stop executing when its signal handler executes and only run again when the signal handler completes (imagine multi processors/cores).

89 [T / F] ... 89

90 [T / F] ... 90