#uni/semester3/Betriebssysteme/chapter5/process-api

HOW TO CREATE AND CONTROL PROCESSES?

The **fork()** system call is used to create a new process.

Figure 5.1: Calling fork() (p1.c)

➡ fork() creates an almost identical process, to the OS it looks like two copies of the program running

➡ the new process (child) doesn't start at main(), rather it comes to life as the fork() had been called

➡ child isn't an exact copy

➡ parent gets the child's PID from fork(), but the child gets a return code of zero

➡ output is not **deterministic**, sometimes child process might say its text first

➡ the CPU **scheduler** determines which process runs

Figure 5.2: Calling fork() And wait() (p2.c)

➡ with system call **wait()** the parent, if first, will wait for child process to finish, before continuing

➡ wait(NULL) (aka **waitpid(-1, NULL, 0);**) waits for all background jobs to finish

➡ maybe better said, wait returns when child process changed state

Figure 5.3: Calling fork(), wait(), And exec() (p3.c)

➡ fork() is useful if you want to run copies of the same program

➡ exec() is to run a different program (there is an exec methods family, each with different parameters). It loads code and static data from the executable **wc** and overwrites its current code segment. Heap and stack are being reinitialised. The OS then doesn't create a new process, rather it transforms the currently running program (p3) into a different program (wc).

➡ if exec() successful, it never returns

➡ separation between fork() and exec() is essential, because it lets you run code after fork but before exec. This code can be used to alter the environment for the next program

Figure 5.4: All Of The Above With Redirection (p4.c)

➡ the child process closes the default output descriptor (STDOUT_FILENO) and opens a file

➡ subsequent writes by child process to the output file descriptor will be redirected to that file (that is why maybe exec doesn't create new process, but transforms the current one)

➡ UNIX pipes are implemented in a similar way. Output of one process is connected to a pipe and the input of another process is connected to the same pipe

➡ **fork and exec combination is a powerful way to create and manipulate processes**

Process control

- ➡ kill() sends signal to process
- ➡ ctrl-c sends SIGINT (interrupt) signal
- ➡ ctrl-z sends SIGTSTP (stop) stop
- ➡ there are ways to send signals individually as well as entire process groups (signal() is used to catch various signals)
- ➡ who is allowed to send signals to processes? Usually users can only control their own processes

Read summary!

Process-Api script

- ➡ in parent process, the string is written to the output
- ➡ in child process, that written string is read, and put in readbuffer