Topic 1: Process Management

---

fork

Declaration: `pid_t fork();`
Usage/Explanation: Copies the parent process to the child process and returns with the two processes.

Same code, Same memory, Same environment (files, etc.). But they are separate processes with separate memory spaces and different PIDs.
Parameters: None
Return Values: Returns 0 to child, PID of child to parent, -1 on error.
Additional:

---

wait

Declaration: `pid_t wait(int *wstatus);`
Usage/Explanation: Waits until **any** child process ends, suspending the calling process until a child terminates
Parameters: `wstatus`: Pointer to the variable where the exit status of the child will be stored. If NULL, no status is returned.

To get the value of the status, you can use macros like `WEXITSTATUS(*wstatus)` which return the second byte of the variable, where the exit code of the child is stored.
Return Values: If there are no children or all children have already terminated and waited for, it returns -1. Else waits until a child process ends and returns its PID.
Additional: Can only wait for direct child!

---

waitpid

Declaration: `pid_t waitpid(pid_t pid, int *wstatus, int options);`
Usage/Explanation: Wait until a specific child process ends.
Parameters:

- `pid`: The PID of the child process to wait for. If -1, waits for any child process.

- `wstatus`: Pointer to an integer where the exit status of the child will be stored. If NULL, no status is returned.
- `options`: Options for waiting, such as WNOHANG (do not block if no child has exited). default is 0, which blocks until the child exits.

Return Values: Returns the PID of the child that exited, or -1 on error. If WNOHANG is set and no child has exited, it returns 0.
Additional:

---

exit

Declaration: `void exit(int status);`
Usage/Explanation: Terminates the calling process and releases all of its recourses. The process becomes *zombie* until its parent process requests to check its termination (e.g. wait()) and then clears completely.
Parameters: `status`: The exit status of the process which is returned to the parent process when checked.
Return Values: No return value. The process is terminated immediately, and will never fail.
Additional: The `main()` is not in fact the main function of the process, it is wrapped by `int __libc_start_main()` who collects the return value of `main()` and calls `exit()` with it. This is why we don't use `exit()` in `main()` but rather return from it usually $\implies$ `exit` is always called.

---

execv

Declaration: `int execv(const char *filename, char *const argv[]);`
Usage/Explanation: Replaces the current running process code with a new program. (same PID,PPID but different code and memory).
Parameters:

- `filename`: The path to the file containing the program to execute.
- `argv`: An array of pointers to null-terminated strings containing the parameters to pass to the new program. The first element is the name of the process, i.e. `argv[0]=filename`. The last argument must be `NULL` to indicate the end of the array.

Return Values: Returns -1 on error, and does not return on success as the current process is replaced by the new program.
Additional: The `execv()` function is one of the *exec* family of functions, which replace the current process image with a new process image. It does not create a new process; it replaces the current one. The `v` stands for the array of arguments, `p` is for searching in the `PATH` environment variable for the filename.

getpid, getppid

Declaration: `pid_t getpid();`
`pid_t getppid();`
Usage/Explanation: `getpid()` returns the PID of the calling process, and `getppid()` returns the PID of the parent process (not real parent).
Parameters: None
Return Values: Returns the PID of the calling process or its parent accordingly.
Additional:

Topic 2: IPC & I/O

---

kill

Declaration: `int kill(pid_t pid, int sig);`
Usage/Explanation: sends signam num. `sig` to the process with PID `pid`.
Parameters:

- `pid`: The PID of the process to send the signal to.
  If `pid` is 0, the signal is sent to all processes in the same process group as the calling process.

- `sig`: The signal number to send. (e.g. `SIGKILL`, `SIGTERM`, etc.).

Return Values: Returns 0 on success, -1 on error. e.g. if the process doesn't exist
Additional: Since there is no signal with the number 0, it is used to check if the process exists or not. If the process exists, it returns 0, else it returns -1. (e.g. `kill(<pid>,0)`).

---

signal

Declaration: `sighandler_t signal(int signum, sighandler_t handler);`
`typedef sighandler_t void (*sighandler_t)(int);`
Usage/Explanation: Sets a signal handler for the specified signal `signum`.
Parameters:

- `signum`: The signal number to set the handler for. (e.g. `SIGINT`, `SIGTERM`, etc.).

- `handler`: The function to call when the signal is received. If `handler` is `SIG_IGN`, the signal is ignored. If `handler` is `SIG_DFL`, the default action for the signal is restored.

Return Values: On success, returns the previous signal handler for the specified signal. If there was no previous handler, it returns `SIG_DFL` or `SIG_IGN`. On error, it returns `SIG_ERR`.
Additional:

---

open

Declaration: `int open(const char *pathname, int flags, mode_t mode);` (mode_t is optional)
Usage/Explanation: Opens the requested file by `pathname` for access with the properties specified by `flags` and permissions specified by `mode`.
Parameters:

- `pathname`: The path to the file (or device) to open.
- `flags`: Flags that specify how the file should be opened. Must include one of the following:
  - `O_RDONLY`: Open for reading only.
  - `O_WRONLY`: Open for writing only.
  - `O_RDWR`: Open for reading and writing.

  Additional flags can be combined using the bitwise OR operator (`|`), such as:
  - `O_CREAT`: Create the file if it does not exist.
  - `O_TRUNC`: Truncate the file to zero length if it already exists.
  - `O_APPEND`: Append data to the end of the file.
- `mode`: Optional parameter that specifies the permissions of the file if it is created. It is used only if the `O_CREAT` flag is set, and is a must! It is a bitwise OR of the following permission bits:
  - `S_IRUSR`: Read permission for the owner.
  - `S_IWUSR`: Write permission for the owner.
  - `S_IXUSR`: Execute permission for the owner.
  - `S_IRGRP`: Read permission for the group.
  - `S_IWGRP`: Write permission for the group.
  - `S_IXGRP`: Execute permission for the group.
  - `S_IROTH`: Read permission for others.
  - `S_IWOTH`: Write permission for others.
  - `S_IXOTH`: Execute permission for others.

Return Values: In case of success, returns a file descriptor (an integer) that refers to the opened file. The given FD is the lowest available index in the FDT of the process. If the file cannot be opened, it returns -1 and sets `errno` to indicate the error.
Additional:

close

Declaration: `int close(int fd);`
Usage/Explanation: Closes the file descriptor `fd`, releasing the resources associated with it.
Parameters: `fd`: The file descriptor to close. It must be a valid file descriptor that was previously opened using `open()` or similar functions.
Return Values: Returns 0 on success, or -1 on error. If the file descriptor is invalid or already closed, it returns -1 and sets `errno` to indicate the error.
Additional: After closing an FD, you can no longer use it to access the file.

---

read

Declaration: `ssize_t read(int fd, void *buf, size_t count);`
Usage/Explanation: Reads up to `count` bytes from the file descriptor `fd` into the buffer pointed to by `buf`.
Parameters:

- `fd`: The file descriptor to read from.
- `buf`: A pointer to the buffer where the read data will be stored.
- `count`: The maximum number of bytes to read from the file descriptor.

Return Values: In case of success, returns the number of bytes read (which can be less than `count` if fewer bytes are available). If the end of the file is reached, it returns 0. On error, it returns -1 and sets `errno` to indicate the error.
Additional: `read()` is a blocking call by default, meaning it will wait until data is available to read.
Note that the seek pointer of the `fd` is advanced by the number of bytes read, so subsequent reads will continue from where the last read left off.

---

write

Declaration: `ssize_t write(int fd, const void *buf, size_t count);`
Usage/Explanation: Writes up to `count` bytes from the buffer pointed to by `buf` to the file descriptor `fd`.
Parameters:

- `fd`: The file descriptor to write to.
- `buf`: A pointer to the buffer containing the data to write.
- `count`: The number of bytes to write from the buffer.

Return Values: In case of success, returns the number of bytes written (which can be less than `count` if fewer bytes can be written). On error, it returns -1 and sets `errno` to indicate the error.
Additional: As with `read()`, `write()` is a blocking call by default, meaning it will wait until the data can be written. And the seek pointer of the `fd` is advanced by the number of bytes written.

---

pipe

Declaration: `int pipe(int filedes[2]);`
Usage/Explanation: Creates a unidirectional data channel *(pipe)* with two FDs: one for reading and one for writing.
Parameters:

- `filedes`: An array of two integers that the syscall will fill with:

  1. `filedes[0]`: The file descriptor for reading from the pipe.
  2. `filedes[1]`: The file descriptor for writing to the pipe.

Return Values: Return 0 if successful, otherwise -1.
Additional:

---

dup, dup2

Declaration: `int dup(int oldfd);`
`int dup2(int oldfd, int newfd);`
Usage/Explanation: Creates a copy of the file descriptor `oldfd` and returns a new file descriptor that refers to the same open file description.
Parameters:

- `oldfd`: The file descriptor to duplicate. Must be an open file descriptor.

- `newfd`: The desired new file descriptor (only for `dup2()`). If `newfd` is already open, it will be closed before being reused.

Return Values: For `dup()`: Returns the lowest numbered unused file descriptor.
For `dup2()`: Returns `newfd` if successful, or -1 on error.
Additional: If we want to close a file we need to close all of its FDs, which they all point to the same *File Object*

---

mkfifo

Declaration: `int mkfifo(const char *pathname, mode_t mode);`
Usage/Explanation: Creates a named pipe (FIFO) with the specified `pathname` and permissions `mode`.
Parameters:

- `pathname`: The path where the FIFO will be created.
- `mode`: The permissions for the FIFO, similar to those used in the `open()` syscall. 0777 means XRW for all users.

Return Values: 0 on success, -1 on error.
Additional: FIFO are shown as files in the filesystem, but they are not saved on disk. It is a bidirectional communication channel between processes.
Note: The FIFO must be opened by at least one process before any other process can write to it, i.e. it's blocking.
If opened for read & write it will be non-blocking