Inter-process communications (IPC)

Week 04 – Lab 4

https://goo.gl/6hsOK6

Means of IPC

- Pipes
- Signals
- waitpid()
- exit()

Pipes

$sort <file.txt | head

- *sort*'s standard output is connected to *head*'s standard input
- all the data that *sort* writes go directly to *head*, instead of going to a file
- If the pipe fills, the system stops running sort until head has removed some data from it

pipe()

- System call to create unnamed pipe
- Traditional pipes implementation uses filesystem to transfer data between processes
- After opening a pipe, regular file related system calls can be used - read(), write(), close() etc.

Exercice 1

- Compile and run the following source code

```
#include <stdio.h>

int main() {
        char* string = "Hello";
        char buf[1024];
        int fds[2];

        pipe(fds);
        printf("buf content before: %s\n", buf);
        write(fds[1], string, 6);
        read(fds[0], buf, 6);
        printf("buf content after: %s\n", buf);
}
```

Exercise 2

- Write a program that creates a child process and transfers the string “Hello World” from the parent process to the child process using pipes. Child process should print the string after receiving it from the pipe.

Signals

- Signals - software interrupts
- Process is responsible for handling incoming signals
- Process can ignore, catch and handle or terminate upon signal arrival
- Default behaviour - terminate
- Process can send signals only to members of it's **process group** (process hierarchy)

Exercise 3

- Compile and run the following program. Press Ctrl+C after execution of ex3 and see what happens.

```
#include<stdio.h>
#include<signal.h>

void sig_handler(int signo) {
        if (signo == SIGINT) {
                printf("received SIGINT\n");
        }
}

int main(void) {
        int i = 0;
        if (signal(SIGINT, sig_handler) == SIG_ERR) {
                printf("\ncan't catch SIGINT\n");
        }
        while(i < 10) {
                sleep(1);
                i = i + 1;
        }
        return 0;
}
```

Linux signals

Name	Number	Description
SIGHUP	1	Hangup (POSIX)
SIGINT	**2**	**Terminal interrupt (ANSI)**
SIGQUIT	3	Terminal quit (POSIX)
SIGILL	4	Illegal instruction (ANSI)
SIGTRAP	5	Trace trap (POSIX)
SIGIOT	6	IOT Trap (4.2 BSD)
SIGBUS	7	BUS error (4.2 BSD)
SIGFPE	8	Floating point exception (ANSI)
SIGKILL	**9**	**Kill(can't be caught or ignored) (POSIX)**
SIGUSR1	10	User defined signal 1 (POSIX)
SIGSEGV	**11**	**Invalid memory segment access (ANSI)**
SIGUSR2	12	User defined signal 2 (POSIX)
SIGPIPE	**13**	**Write on a pipe with no reader, Broken pipe (POSIX)**
SIGALRM	**14**	**Alarm clock (POSIX)**
SIGTERM	**15**	**Termination (ANSI)**
SIGSTKFLT	16	Stack fault
SIGCHLD	17	Child process has stopped or exited, changed (POSIX)
SIGCONT	18	Continue executing, if stopped (POSIX)
SIGSTOP	19	Stop executing(can't be caught or ignored) (POSIX)

SIGINT, SIGQUIT, SIGTERM, SIGKILL, SIGSTOP

- SIGINT is the interrupt signal. The terminal sends it to the foreground process when the user presses ctrl-c. The default behavior is to terminate the process, but it can be caught or ignored. The intention is to provide a mechanism for an orderly, graceful shutdown.
- SIGQUIT is the dump core signal. The terminal sends it to the foreground process when the user presses ctrl-\. The default behavior is to terminate the process and dump core, but it can be caught or ignored. The intention is to provide a mechanism for the user to abort the process.
- SIGTERM is the termination signal. The default behavior is to terminate the process, but it also can be caught or ignored. The intention is to kill the process, gracefully or not, but to first allow it a chance to cleanup.
- SIGKILL is the kill signal. The only behavior is to kill the process, immediately. As the process cannot catch the signal, it cannot cleanup, and thus this is a signal of last resort.
- SIGSTOP is the pause signal. The only behavior is to pause the process; the signal cannot be caught or ignored. The shell uses pausing (and its counterpart, resuming via SIGCONT) to implement job control.

Exercise 4

- Write a C program that declares signal handlers for SIGKILL, SIGSTOP and SIGUSR1 (refer to Exercise 3). Compile and run the program in the background (./ex4 &). Use kill command to send USR1 signal to the process you have started: $kill -USR1 <pid>. Explain the output

Exercise 5

- Write a C program that fork's a child process, waits for 10 seconds and then sends a SIGTERM signal to the child. Child process should run an infinite loop and print "I'm alive" every second. Hint: use kill() C function to send a signal.

wait() and waitpid()

- The wait() system call suspends execution of the current process until one of its children terminates
- The waitpid() system call suspends execution of the current process until a child specified by pid argument has changed state

waitpid() semantics

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

- Return value - pid of the process that has terminated
- 1st argument: pid of the process to wait on (-1 for any child process). 2nd argument - address of status variable, which will be set according to the exit reason of the terminated process. 3rd parameter - additional options, can be left 0

exit()

- Processes should use exit() when they are finished executing.
- exit() has one parameter, the exit status (0 to 255), which is returned to the parent in the variable status of the waitpid system call.

Exercise 6

- Write a C program that forks a child process, checks on it's status using waitpid() call and exits once the child has terminated. Childs should sleep for 10 seconds and exit.