**42 Common Core | Milestone 2**

**MINITALK**

**Theoretical Overview**

**CONCEPTS**

**UNIX signals:** form of inter-process communication (IPC) used in UNIX systems to notify a process that a specific event has occur.

**SIGUSR1 & SIGUSR2:** user-defined signals intended to be used freely by user programmes. Their default action is to terminate process, but you can assign other meanings to them through a signal handler.

**PID** (short for **Process Identifier**): unique number (positive integers) assigned by the OS to each running process.

**Global variable ‒ volatile sig\_atomic\_t:**

* **sig\_atomic\_t:** integer type defined in signal.h that is **atomic** with respect to signals, i.e., reading or writing this type is a single, indivisible operation, which cannot be interrupted by a signal handler.
* **volatile:** keyword thattells the compiler not to optimize accesses to the variable and read its current value directly from memory, since it can change outside the programme’s usual flow (e.g. by a signal handler or hardware).

**Unicode** (short for **Universal Character Encoding Standard**): supports all human-readable characters, covering virtually all writing systems, plus symbols and emojis. It has over 1.1 million code points (with ASCII covering only the first 128).

* **UTF-8** (short for **Unicode Transformation Format – 8-bit):** most common character encoding that represents each Unicode character as a valid, self-delimiting byte sequence (1 to 4 bytes). The first byte of the sequence tells how many total bytes follow.

The **main challenge** of the project is to ensure both:

* **signal integrity** (no lost or corrupted data): the server’s behaviour depends on timing, so too-fast signalling from the client can result in dropped signals or out-of-order interpretation, leading to incomplete or incorrect bytes.
* **robust and swift response:** the server needs to handle input from multiple clients in sequence without crashing, freezing, or leaking memory, and do it fast (over 100 characters per second).

**FUNCTIONS**

**sigaction**

|  |  |
| --- | --- |
| Prototype | #include <signal.h>  int sigaction(int signum, const struct sigaction \*act, struct sigaction \*oldact); |
| Purpose | Modern way to handle signals.  Compared to **signal**, it is more reliable, safer, gives you access to more detailed info (like sender PID), and more control.  It installs a struct sigaction, which specifies:   * handler function: sa\_handler – simple handler OR sa\_sigaction – extended handler that receives extra info; * sa\_flags: options to modify behaviour, like SA\_SIGINFO to use sa\_sigaction instead of sa\_handler; * sa\_mask (signal mask): allows you to block other signals while handler runs. |
| Parameters | int signum: signal number.  const struct sigaction \*act: pointer to sigaction struct specifying the new action for the signal; if NULL, no changes are made.  struct sigaction \*oldact: pointer to a sigaction struct where the previous action is saved; if NULL, it doesn’t save the old action. |
| Return value | 0 on success or -1 on failure |

**sigemptyset**

|  |  |
| --- | --- |
| Prototype | #include <signal.h>  int sigemptyset(sigset\_t \*set); |
| Purpose | Initializes a signal set to empty, i.e., to contain no signals. This prevents signals from being unintentionally blocked.  Commonly used in conjunction with **sigemptyset**, which adds signals to a signal set. |
| Parameters | sigset\_t \*set: pointer to a signal set object to initialize to empty. |
| Return value | 0 on success or -1 on failure |

**kill**

|  |  |
| --- | --- |
| Prototype | #include <signal.h>  int kill(pid\_t pid, int sig); |
| Purpose | Sends any specified signal to a process or group of processes. |
| Parameters | pid\_t pid: if pid > 0, sends signal to the process with this PID; if pid == 0, sends signal to processes in the same process group as the sender; if pid == -1, sends signal to all processes for which the sender has permission (except process 1), if pid < -1: sends signal to all processes in the group -pid.  int sig: if sig == 0, no signal is sent but it performs error check (whether a process exist and is accessible). |
| Return value | 0 on success or -1 on failure |

**getpid**

|  |  |
| --- | --- |
| Prototype | #include <unistd.h>  pid\_t getpid(void); |
| Purpose | Returns the process ID (PID) of the calling process. |
| Parameters | none |
| Return value | PID (always successful) |

**usleep**

|  |  |  |
| --- | --- | --- |
| Prototype | #include <unistd.h>   |  | | --- | | int usleep(useconds\_t usec); | |
| Purpose | Suspends execution for microsecond intervals (usec). |
| Parameters | useconds\_t usec: number of microseconds to sleep. |
| Return value | 0 on success and -1 on failure |

**pause**

|  |  |
| --- | --- |
| Prototype | #include <unistd.h>  int pause(void); |
| Purpose | Suspends the calling process until a signal is received. |
| Parameters | none |
| Return value | -1 |

**write**

|  |  |
| --- | --- |
| Prototype | #include <unistd.h>  ssize\_t write(int fd, const void \*buf, size\_t count); |
| Purpose | Writes up to count bytes from buf to the file descriptor fd. |
| Parameters | fd: file descriptor (e.g., 1 for stdout, 2 for stderr)  buf: pointer to the data buffer to write  count: number of bytes to write |
| Return value | number of bytes written on success or -1 on failure |