```
// stringi
// split po delimiterach, nastepne wywolania z NULLEM
// zwraca ptr na part lub NULL jak string sie skonczyl
strtok(str, ' ')
strtok(NULL, ' ')
strdup(str) //duplikuje stringa
strcmp(str1, "asdf") //zero jak rowne!


// pliki biblioteczne
fopen(tempFile, "r")
fread(buffer, sizeof(char), ile, fp); // jak chcemy bajty wczytac // 0 jak juz nic nie ma
fgets(buffer, sizeof(buffer), fp); // wczytuje do znaku nowej linii // NULL jak nie ma juz co czytac
fwrite(buffer, sizeof(char), size, f); //wpisuje bajty do pliku
fputs(str, fp) // wpisuje string
fprintf(format, str, fp) // wpisuje string ale z formatem
fseek(fp, 0, SEEK_END);
rewind(fp); //przewin plik


// pliki systemowe
int fd = open(filename, O_RDWR) // O_WRONLY | O_CREAT | O_TRUNC | O_RDONLY | O_APPEND
int creat(const char *pathname, mode_t mode);
read(f, buffer, size); // 0 jak juz nic nie ma
lseek(f, line*size, 0);
write(f, buffer, size);
close(fd)


// shared libki
dlopen("../zad1/diffLib.so", RTLD_LAZY);
void (*func)(int) = (void (*)(int)) dlsym(handle, "create_main_block"); //handle z dlopen


// czasy
time(NULL) // returns the time as the number of seconds since the Epoch, 1970-01-01 00:00:00 +0000 (UTC).
double difftime(time_t time1, time_t time0); //roznica czasow, w double ^

clock_t clock_t_begin, clock_t_end;
struct tms times_start_buffer, times_end_buffer;

void start_timer(){
        clock_t_begin = times(&times_start_buffer);
}

void stop_timer(){
        clock_t_end = times(&times_end_buffer);
}

double calc_time(clock_t s, clock_t e) {
        return ((long int) (e - s) / (double) sysconf(_SC_CLK_TCK));
}

void print_times(const char* operation){
        printf("%20s   real %.3fs   user %.3fs   sys %.3fs\n",
                    operation,
                    calc_time(clock_t_begin, clock_t_end),
                    calc_time(times_start_buffer.tms_cutime, times_end_buffer.tms_cutime),
                    calc_time(times_start_buffer.tms_cstime, times_end_buffer.tms_cstime));
}

// filesystem
const char* get_file_type(mode_t m){
        if (S_ISREG(m)) return "file";
        if (S_ISFIFO(m)) return "fifo";
        if (S_ISDIR(m)) return "dir";
        if (S_ISCHR(m)) return "char dev";
        if (S_ISLNK(m)) return "slink";
        if (S_ISBLK(m)) return "block dev";
        if (S_ISSOCK(m)) return "sock";
        return "unknown";
}

DIR* opendir(const char* dirname);
int closedir(DIR* dirp);
struct dirent* readdir(DIR* dirp)
int stat (const char *path, struct stat *buf);
int lstat(const char *path, struct stat *buf); //identyczne jak stat tylko jak ma linka to zwraca dla linku
```

samego w sobie

```
struct stat {
        dev_t     st_dev;         /* ID of device containing file */
        ino_t     st_ino;         /* Inode number */
        mode_t    st_mode;        /* File type and mode */
        nlink_t   st_nlink;       /* Number of hard links */
        uid_t     st_uid;         /* User ID of owner */
        gid_t     st_gid;         /* Group ID of owner */
        dev_t     st_rdev;        /* Device ID (if special file) */
        off_t     st_size;        /* Total size, in bytes */
        blksize_t st_blksize;     /* Block size for filesystem I/O */
        blkcnt_t  st_blocks;      /* Number of 512B blocks allocated */

        struct timespec st_atim;  /* Time of last access */
        struct timespec st_mtim;  /* Time of last modification */
        struct timespec st_ctim;  /* Time of last status change */
};

int mkdir (const char *path, mode_t mode); //0777
int rmdir (const char *path);
int chmod (const char *path, mode_t new_mode);
//przechodzi po katalogu fn - to handler
int nftw(const char *dir, int(*fn) (const char *, const struct stat *, int, struct FTW *), int nopend, int f
lags);


// procesy
pid_t getpid(void) // zwraca PID procesu wywołującego funkcję
pid_t getppid(void) // zwraca PID procesu macierzystego
uid_t getuid(void) // zwraca rzeczywisty identyfikator użytkownika UID
uid_t geteuid(void) // zwraca efektywny identyfikator użytkownika UID
gid_t getgid(void) // zwraca rzeczywisty identyfikator grupy GID
gid_t getegid(void) // zwraca efektywny identyfikator grupy GID
pid_t fork() // zwraca pid, jesli zero to jestesmy w dziecku, jesli > 0 to w rodzicu else error
int execl(char const *path, char const *arg0, ...) //funkcja jako pierwszy argument przyjmuje ścieżkę do pli
ku, następne są argumenty wywołania funkcji, gdzie arg0 jest nazwą programu
int execle(char const *path, char const *arg0, ..., char const * const *envp) //podobnie jak execl, ale pozw
ala na podanie w ostatnim argumencie tablicy ze zmiennymi środowiskowymi
int execlp(char const *file, char const *arg0, ...) //również przyjmuje listę argumentów ale, nie podajemy t
utaj ścieżki do pliku, lecz samą jego nazwę, zmienna środowiskowa PATH zostanie przeszukana w celu zlokalizo
wania pliku
int execv(char const *path, char const * const * argv) //analogicznie do execl, ale argumenty podawane są w
tablicy
int execve(char const *path, char const * const *argv, char const * const *envp) //analogicznie do execle, r
ównież argumenty przekazujemy tutaj w tablicy tablic znakowych
int execvp(char const *file, char const * const *argv) //analogicznie do execlp, argumenty w tablicy

pid_t wait ( int *statloc ); //czeka na dowolne dziecko BLOKUJE
pid_t waitpid(child_pid, &status, 0); //czeka na pida, ostatni param to opcje, opcja moze nie blokowac WNOHA
NG
// returns 0 on success or if WNOHANG was specified and no child(ren) specified by id has yet changed state

int kill(pid_t pid, int sig);
int raise( int signal); // jak kill tylko w samego siebie
int fno = fileno(fp);
flock(fno, LOCK_EX);
flock(fno, LOCK_UN);

// limity
void set_limits(int cpu, int mem){
        struct rlimit proc = {cpu, cpu};
        struct rlimit memory = {mem * 1000000, mem * 1000000};

        setrlimit(RLIMIT_CPU, &proc);
        setrlimit(RLIMIT_AS, &memory);
}

struct rusage start, end;
getrusage(RUSAGE_CHILDREN, &start);
void print_diff(struct rusage *start, struct rusage *end){
        long u_s =      abs(end->ru_utime.tv_sec -      start->ru_utime.tv_sec);
        long u_us = abs(end->ru_utime.tv_usec - start->ru_utime.tv_usec);
        long s_s =      abs(end->ru_stime.tv_sec -      start->ru_stime.tv_sec);
        long s_us = abs(end->ru_stime.tv_usec - start->ru_stime.tv_usec);

        printf("user time: %ld.%04lds\n", u_s, u_us);
        printf("sys  time: %ld.%04lds\n", s_s, s_us);
}
```

```c
//sygnaly
int kill(pid_t pid, int sig);
int raise( int signal); // jak kill tylko w samego siebie
int sigqueue(pid_t pid, int sig, const union sigval value);
union sigval {
        int sival_int;
        void *sival_ptr;
};

signal(SIGINT, sigint_handler);
void sigint_handler(int signo);

struct sigaction action;
action.sa_handler = sigtstp_handler;
sigemptyset(&action.sa_mask);
sigaddset(&block_mask, SIGUSR1);
sigaction(SIGTSTP, &action, NULL)

action.sa_flags = 0; //handler bez dodatkowych info
void sigusr2_handler(int signo)

action.sa_flags = SA_SIGINFO; // jesli ustawimy ta flage to
void sigfpe_handler(int signo, siginfo_t *info, void *context)

siginfo_t {
        int      si_signo;     /* Signal number */
        int      si_errno;     /* An errno value */
        int      si_code;      /* Signal code */
        int      si_trapno;    /* Trap number that caused hardware-generated signal (unused on most architec
tures) */
        pid_t    si_pid;       /* Sending process ID */
        uid_t    si_uid;       /* Real user ID of sending process */
        int      si_status;    /* Exit value or signal */
        clock_t  si_utime;     /* User time consumed */
        clock_t  si_stime;     /* System time consumed */
        sigval_t si_value;     /* Signal value */
        int      si_int;       /* POSIX.1b signal */
        void    *si_ptr;       /* POSIX.1b signal */
        int      si_overrun;   /* Timer overrun count; POSIX.1b timers */
        int      si_timerid;   /* Timer ID; POSIX.1b timers */
        void    *si_addr;      /* Memory location which caused fault */
        long     si_band;      /* Band event (was int in glibc 2.3.2 and earlier) */
        int      si_fd;        /* File descriptor */
        short    si_addr_lsb;  /* Least significant bit of address (since Linux 2.6.32) */
        void    *si_lower;     /* Lower bound when address violation occurred (since Linux 3.19) */
        void    *si_upper;     /* Upper bound when address violation occurred (since Linux 3.19) */
        int      si_pkey;      /* Protection key on PTE that caused fault (since Linux 4.6) */
        void    *si_call_addr; /* Address of system call instruction (since Linux 3.5) */
        int      si_syscall;   /* Number of attempted system call (since Linux 3.5) */
        unsigned int si_arch;  /* Architecture of attempted system call (since Linux 3.5) */
}



sigset_t block_mask;
sigemptyset(&block_mask);
sigaddset(&block_mask, SIGUSR1);
sigprocmask(SIG_BLOCK, &block_mask, NULL); //maskujemy sygnaly
sigfillset(&block_mask); //wypelnia maske, wszystko blokuje
sigdelset(&block_mask, signal); //usuwa sygnal ze zbioru

sigset_t current_signals;
sigpending(&current_signals); //pobiera oczekujace sygnaly
sigismember(&current_signals, SIGUSR1) //sprawdza czy w obecnej strukturze jest podany sygnal


// procesy
int fd[2];
pipe(fd) //tworzymy pipe [0] - read, [1] - write
int dup2(int oldfd, int newfd); //kopiuje oldfd do newfd,
FILE *f = popen("asdf", "r"); //tworzy potok, proces, ustawia jego stdin lub stdout na stosowną końcówkę pot
oku, 'r' czytamy wyjscie, 'w' piszemy do
pclose(f);

int mkfifo(const char *pathname, 0666); //tworzy potok nazwany
int mknod(const char *pathname, mode_t mode, dev_t dev);
//na fifo mozna wolac normalne fopen, open itp
```