unistd.h

fcntl.h
ioctl.h

STDIN_FILENO
: File number of *stdin*. It is 0.

STDOUT_FILENO
: File number of *stdout*. It is 1.

STDERR_FILENO
: File number of *stderr*. It is 2.

Type Definitions

The **size_t**, **ssize_t**, **uid_t**, **gid_t**, **off_t** and **pid_t** types are defined as described in <sys/types.h>.

The **useconds_t** type is defined as described in <sys/types.h>.

The **intptr_t** type is defined as described in <inttypes.h>.

Declarations

The following are declared as functions and may also be defined as macros. Function prototypes must be provided for use with an ISO C compiler.

```
int          access(const char *, int);
unsigned int alarm(unsigned int);
int          brk(void *);
int          chdir(const char *);
int          chroot(const char *); (LEGACY)
int          chown(const char *, uid_t, gid_t);
int          close(int);
size_t       confstr(int, char *, size_t);
char        *crypt(const char *, const char *);
char        *ctermid(char *);
char        *cuserid(char *s); (LEGACY)
int          dup(int);
int          dup2(int, int);
void         encrypt(char[64], int);
int          execl(const char *, const char *, ...);
int          execle(const char *, const char *, ...);
int          execlp(const char *, const char *, ...);
int          execv(const char *, char *const []);
int          execve(const char *, char *const [], char *const []);
int          execvp(const char *, char *const []);
void        _exit(int);
int          fchown(int, uid_t, gid_t);
int          fchdir(int);
int          fdatasync(int);
pid_t        fork(void);
long int     fpathconf(int, int);
int          fsync(int);
int          ftruncate(int, off_t);
char        *getcwd(char *, size_t);
int          getdtablesize(void); (LEGACY)
gid_t        getegid(void);
uid_t        geteuid(void);
gid_t        getgid(void);
int          getgroups(int, gid_t []);
long         gethostid(void);
char        *getlogin(void);
int          getlogin_r(char *, size_t);
int          getopt(int, char * const [], const char *);
int          getpagesize(void); (LEGACY)
char        *getpass(const char *); (LEGACY)
pid_t        getpgid(pid_t);
pid_t        getpgrp(void);
pid_t        getpid(void);
pid_t          getppid(void);
pid_t          getsid(pid_t);
uid_t          getuid(void);
char          *getwd(char *);
int            isatty(int);
int            lchown(const char *, uid_t, gid_t);
int            link(const char *, const char *);
int            lockf(int, int, off_t);
off_t          lseek(int, off_t, int);
int            nice(int);
long int       pathconf(const char *, int);
int            pause(void);
int            pipe(int [2]);
ssize_t        pread(int, void *, size_t, off_t);
int            pthread_atfork(void (*)(void), void (*)(void),
                   void(*)(void));
ssize_t        pwrite(int, const void *, size_t, off_t);
ssize_t        read(int, void *, size_t);
int            readlink(const char *, char *, size_t);
int            rmdir(const char *);
void          *sbrk(intptr_t);
int            setgid(gid_t);
int            setpgid(pid_t, pid_t);
pid_t          setpgrp(void);
int            setregid(gid_t, gid_t);
int            setreuid(uid_t, uid_t);
pid_t          setsid(void);
int            setuid(uid_t);
unsigned int   sleep(unsigned int);
void           swab(const void *, void *, ssize_t);
int            symlink(const char *, const char *);
void           sync(void);
long int       sysconf(int);
pid_t          tcgetpgrp(int);
int            tcsetpgrp(int, pid_t);
int            truncate(const char *, off_t);
char          *ttyname(int);
int            ttyname_r(int, char *, size_t);
useconds_t     ualarm(useconds_t, useconds_t);
int            unlink(const char *);
int            usleep(useconds_t);
pid_t          vfork(void);
ssize_t        write(int, const void *, size_t);
```

The following external variables are declared:

```
extern char   *optarg;
extern int    optind, opterr, optopt;
```

APPLICATION USAGE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

access(), alarm(), chdir(), chown(), close(), crypt(), ctermid(), dup(), encrypt(), environ(), exec, exit(), fchdir(), fchown(), fcntl(), fork(), fpathconf(), fsync(), ftruncate(), getcwd(), getegid(), geteuid(), getgid(), getgroups(), gethostid(), getlogin(), getpgid(), getpgrp(), getpid(), getppid(),

Unistd.h

void exit(int status)

- It ends a process and returns a value to its parent.
- status is an integer between 0 to 255. This number is returned to the parent via wait() as the exit status of the process.

exit() is a <stdlib.h> library function that calls the system call _exit(). exit() cleans up the standard I/O streams (tmpfile()) before calling _exit().

- Calling _exit() instead of exit() will bypass this cleanup procedure.
- wait() returns the exit status multiplied by 256 (stored in upper 8 bits). The status value is shifted right 8 bit (divided by 256) to obtain the correct value.

int chown(const char *path, int uid owner, int gid group);

- It sets the owner ID and grup ID of the file specified by path to owner & group respectively.
- If owner or group ID is specified as -1 or -1 respectively the corresponding ID of file is unchanged.
- It returns 0 upon successful completion, otherwise -1 is returned.
- It fails if searching permission is denied.

int fchown(int fd, uid, gid);

- fchown() has the same effect as chown() except that the file path and file description parameter.

Unistd.h   Unix Standard

pid_t fork( );

P ← child's PID

C ← 0

-1 → if Error ⟹ No Resources, insufficient memory and CHILD_MAX

* pid_t getpid(); Return pid of calling process.

No return value is reserved to indicate an error.

* pid_t getppid(): Return parent process ID of calling Process

No any error

→ If called in parent process, it returns shell's process id no.

* int nice(int incr): Return new nice value, -1 in error

→ adds the value of incr to the nice value of the calling process.

→ nice value is a non-negative number.

→ Minimum nice value of 0 (zero) are imposed by system

→ No effect on SCHED_FIFO & SCHED_RR

→ If the process is multi-threaded, the nice value affects all system scope threads in the process.

→ more positive value results in a lower CPU priority

* unsigned int sleep(unsigned int seconds);

- Returns the requsted time has elapsed, the value returned will be 0.
- No errors are defined.

* Pid value will always be unique. Once the process finishe execution its pid value is returned to the kernel which ma re-use it for another process sometime later.

→ Pid value can't be changed when assigned during execution.

* fork() duplicates the variable which are before fork(). It also duplicated even the Global variable (declared before main())

* fork() simply shares the files instead of duplicates.

* globally opened file uses the same descriptor. All processes use that one descriptor.

Unistd.h

<signal.h>

int pause(void);   SIGCONT

It suspends the execution of the calling process untill it receives a signal.

- There is no successful completion return value. -1 is returned in case of error.

- It fail if a signal is caught by the calling process and Control is returned from the signal-catching function.

```
              int         char*
int read(fd, buffer-pointer, int nbyte);
int write(fd, buffer-pointer, int nbyte);
```

- fd identifies the I/o channel.

- buffer-pointer points to the area in memory where the data is stored for a read() or where the data is taken for a write().

- nbyte defines the maximum number of characters or bytes transfered between the file and the buffer.

- read() & write() return the number of bytes transfered.

- There is no limit on nbyte size or depends on SSIZE-MAX. (implementation-dependent).

- A nbyte of 1 is used to transfer a byte at a time so called unbuffered I/O.

- most efficient value for nbyte is the size of the largest physical record the I/O channel is likely to have to handle

- read() & write() return a non-negative integer indicating the number of byte actually read or wr

- In case of failure, return -1 & set errno to ind error.

int close(int fd);

- It will deallocate the fd. The deallocated fd will be availa for further use for open() & creat(). All the outstanding record locks owned by the process on the file associated with fd will also be removed.

- It returns 0 on successful completion. Otherwise, it returns -1 and set errno to indicate the error.

int access(const char* path, int amode);

- It checks the file pointed by path argument is accessible according to the amode, using real user ID and real group ID.

- The amode constants are defined in sys/file.h

F_OK	existence
X_OK	execute
W_OK	write
R_OK	read

* The constant values may be ORed to check more than one access permission.

- It returns 0, if requested access is permitted. Otherwise returns -1 to set the errno.

- access() only answer the question "do I have this permission" It can't answer the question "what permission do I have pid.t setsid (void)

It creates a new session, if the calling process is not a process group leader. Upon return the calling process will be the session leader of the new session, will be the process group leader of a new process group, and will have no controlling terminal.

→ The process group ID of the calling process will be set equal to the process ID of the calling process.

→ The calling process will be the only process in the new process group and the only process in the new session.

→ It returns the value of the process group ID of the calling process on successful completion. Otherwise returns -1 & set errno to indicate the error.

- It fails if the calling process is already a process group leader.

int setpgid($\overset{\text{int}}{\text{pid.t}}$ pid, $\overset{\text{int}}{\text{pid.t}}$ pgid)

- It is used either to join an existing process group or create a new process group within the session of the calling process. The process group ID of a session leader will not change. Upon successful completion, the process group ID of the process with a process ID that matches pid will be set to pgid.
- As a special case, if pid is 0, the process ID of the calling process will be used. Also, if pgid is 0, the process group ID of the indicated process will be used.
- Upon successful completion, it returns 0, otherwise -1 is returned and errno is set.
- setpgrp() is equivalent to setpgid (0,0)

$\overset{\text{int}}{\text{pid\_t}}$ getpgid(pid_t $\overset{\text{int}}{\text{pid}}$)

→ It returns the process group ID of the process whose process ID is equal to pid.

→ If pid is equal to 0(zero), it returns the process group ID of the calling process.

- It is fails, it returns or set the errno to indicate the error.

- getpgid() fails if → ① The process whose process ID is equal to pid is not in the same session as the calling process, and the implementation does not allow access to the process group ID of that process from the calling process.
② There is no process with a process ID equal to pid.

Pid_t getpgrp(void)

- It returns the process group ID of the calling process.
- It always successful and no return value is peserved to indicate an error.

* getpgrp() is equivalent to getpgid(0).

- Each process group is a member of a session and each process is a member ~~of~~ the session of which its process group is a member.

Pid_t setpgrp(void);

If the calling process is not already a session leader, setpgrp() sets the process group ID of the calling process to the process ID of the calling process. If setpgrp() creates a new session, then the new session has no controlling terminal.

- setpgrp() has no effect when the calling process is a session leader.
- No any error is defined
- Upon Completion, it returns the process group ID.

int kill (Pid_t pid, int sig)

→ It will send a signal to a process or a group of processes specified by pid. The signal to be sent is specified by sig and is either one from the list given in <signal.h> or 0.

→ If sig is 0 (the null signal), error checking is performed but no signal is actual sent. The null signal can be used to check the validity of pid.

→ If pid is greater than 0, sig will be sent to the process whose process ID is equal to pid.

→ If pid is 0, sig will be sent to all processes whose process group ID is equal to the process group ID of the sender, and for which the process has permission to send a signal.

→ If pid is -1, sig will be sent to all processes for which the process has permission to send that signal.

→ If pid is negative, but not -1, sig will be sent to all processes whose process group ID is equal to the absolu value of pid, and for which the process has permission send a signal.

→ Upon successful it returns 0 and send sig to any of the processes specified by pid.

→ If kill() fails, no signal will be sent, and returns -1, and set errno.

→ Kill() fails if ① the value of the sig is an invalid or unsupported signal number. ② The process does not have permission to send the signal to any receiving process. ③ No process or process group can be found corresponding to that specified by pid.

```
                        #include <unistd.h>
extern char **environ;
int execl(const char *path, const char *arg0, ... /*, (char *)0 */);
int execv(const char *path, char *const argv[]);
int execle(const char *path, const char *arg0, ... /*, (char *)0,
          char *const envp[]*/);
int execve(const char *path, char *const argv[], char *const envp[]);
int execlp(const char *file, const char *arg0, ... /*, (char *)0 */);
int execvp(const char *file, char *const argv[]);
```

- The arguments specified by a program with one of the `exec` functions are passed on to the new process image in the corresponding `main()` arguments.
- The argument `path` points to a pathname that identifies the new process image file.
- The exec functions replace the current process image with a new process image. The new image is constructed from a regular, executable file called the new process image file. There is no return from a successful exec, because the calling process image is overlaid by the new process image.
- When a C-language program is executed as a result of this call, it is entered as a C-language function call as follows:

```
    int main (int argc, char *argv[]);
```

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves.

- In addition, the *`environ`* variable:

```
    extern char **environ;
```

- *`environ`* variable is initialized as a pointer to an array of character pointers to the environment strings.

strings

- The argv and *environ* arrays are each terminated by a null pointer. The null pointer terminating the argv array is not counted in argc.
- The argument file is used to construct a pathname that identifies the new process image file. If the file argument contains a slash character, the file argument is used as the pathname for this file. Otherwise, the path prefix for this file is obtained by a search of the directories passed as the environment variable. If this environment variable is not present, the results of the search are implementation-dependent.
- If the process image file is not a valid executable object, execlp() and execvp() use the contents of that file as standard input to a command interpreter conforming to system(). In this case, the command interpreter becomes the new process image.
- The arguments represented by arg0, ... are pointers to null-terminated character strings. These strings constitute the argument list available to the new process image. The list is terminated by a null pointer. The argument arg0 should point to a filename that is associated with the process being started by one of the exec functions.
- The argument argv is an array of character pointers to null-terminated strings. The last member of this array must be a null pointer. These strings constitute the argument list available to the new process image. The value in argv[0] should point to a filename that is associated with the process being started by one of the exec functions.
- The argument envp is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process image. The envp array is terminated by a null pointer.
- For those forms not containing an envp pointer (eg. execl , execv(), execlp() and execvp()), the environment for the new process image is taken from the external variable environ in the calling process.
- The number of bytes available for the new process' combined argument and environment lists is {ARG_MAX}. It is implementation-dependent whether null terminators, pointers, and/or any alignment bytes are included in this total.

- File descriptors open in the calling process image remain open in the new process image, except for those whose close-on-exec flag FD_CLOEXEC is set. For those file descriptors that remain open, all attributes of the open file description, including file locks remain unchanged.
- Directory streams open in the calling process image are closed in the new process image.
- RETURN VALUE: If one of the exec functions returns to the calling process image, an error has occurred; the return value is -1, and errno is set to indicate the error.
- The new process also inherits at least the following attributes from the calling process image:
    - ✓ nice value (see nice())
    - ✓ semadj values (see semop())
    - ✓ process ID
    - ✓ parent process ID
    - ✓ process group ID
    - ✓ session membership
    - ✓ real user ID
    - ✓ real group ID
    - ✓ supplementary group IDs
    - ✓ time left until an alarm clock signal (see alarm())
    - ✓ current working directory
    - ✓ root directory
    - ✓ file mode creation mask (see umask())
    - ✓ file size limit (see ulimit())
    - ✓ process signal mask (see sigprocmask())
    - ✓ pending signal (see sigpending())
    - ✓ tms_utime, tms_stime, tms_cutime, and tms_cstime (see times())
    - ✓ resource limits

- ✓ controlling terminal
- ✓ interval timers

➢ All other process attributes defined in this document will be the same in the new and old process images. The inheritance of process attributes not defined by this specification is implementation-dependent.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <process.h>
int main(int argc, char* argv[]) {
    char* arr[] = {
        "ping",
        "google.com",
        NULL
    };
    char* env[] = {
        "TEST=environment variables",
        NULL
    };
     // Without env and PATH variable
    _execv("C:\\Windows\\System32\\ping.exe",
          "C:\\Windows\\System32\\ping.exe", "google.com", NULL);
    _execl("C:\\Windows\\System32\\ping", arr);
    // Without env but with PATH variable
    _execvp("ping", "ping", "google.com", NULL);
    _execlp("ping", arr);
    // With env and PATH variable
    _execvpe("ping", "ping", "google.com", NULL, env);
    _execlpe("ping", arr, env);
    return 0;
}
```

* pid_t waitpid(pid_t pid, int * stat_loc, int options)

→ Arg. pid_t pid is greater than 0, It specify the process ID of a single child process whose status is required.

⇒ If pid is 0, status is required for any child process whose process group ID is equal to that of the calling process.

→ If pid is -1, status is requested for any child process. Then waitpid() is equivalent to wait()

Options: WCONTINUED (Ask status)
waitpid() report the status of any continued child process.

WNOHANG (No more wait)
waitpid() shall not suspend the execution of the calling thread if status is not immedialely available for child process.

WUNTRACED
child process is stopped but his status is not yet reported. There status reported to the requesting process.

Zombie :- A process that has finished the execution but still has entry in the process table. example - parent is in suspended or sleep state & child exits.

Orphan :- A process whose parent process no more exists eg. finished or terminated without waiting for its child process to terminate. eg. Parent finished but child still executing sys/wait.h   Synchronization

wait & waitpid - wait for a child process to stop or terminate.

It forces the parent to suspend execution until the child's finished

* pid_t wait(int *stat_loc)

wait returns the process ID of a child process that finished.

stat_loc is a pointer to an integer where value returned by wait is stored.

⇒ wait() fails by two conditions:.

1 The process has no children to wait for

2 stat_loc points to an invalid address.

⇒ Information returned by wait()

1 Ended by exit().

2nd Lowest byte is set to the argument by exit(). Lowest byte is set to zeros

2. Ended by signal.

Lowest byte is set to signal number that ended the process. 2nd Lowest byte is set to zeros