APPENDIX

C

# Kernel API Subset

## IN THIS APPENDIX

This appendix lists all the manual pages of the kernel library and system calls that are not directly related to sockets but are typically used in conjunction with sockets.

# Tasks

Tasks include both processes and threads. Threads (`pThreads`) are defined in the next section; this section covers processes and low-level tasks (clones).

## fork()

Create a new process (independent task) at this call. This call creates a child process to run with the parent. You must be careful that you capture the child and direct it to its assigned task; otherwise, the child runs each statement the parent does (they run together).

### Prototype

```
#include <unistd.h>
pid_t fork(void);
```

### Return Value

| | |
|---|---|
| 0 | The task that gets this is the child. |
| >0 | The task that gets this is the parent. |
| <0 | The parent failed to create a new child; check `errno`. |

### Parameters

(none)

### Possible Errors

| | |
|---|---|
| EAGAIN | The `fork()` cannot allocate sufficient memory to copy the parent's page tables and allocate a task structure for the child. |
| ENOMEM | The `fork()` failed to allocate the necessary kernel structures because memory is tight. |

### Example

```
int PID;
if ( (PID = fork()) == 0 )
{ /*--- CHILD ---*/
    /**** Run the child's assignment ***/
    exit();
}
else if ( PID > 0 )
{ /*--- PARENT ---*/
    int status;
    /**** Do parent's work ****/
    wait(status); /* may be done in SIGCHLD signal handler */
}
else /*--- ERROR ---*/
    perror("fork() failed");
```

## __clone()

This is a low-level system call for creating tasks. You can directly control what is shared between the parent and the child. This is not for amateur programmers; you can create very unpredictable programs. (See Chapter 7, “Dividing the Load: Multitasking,” for a complete description of this call.)

### Prototype

```
#include <sched.h>
int __clone(int (*fn)(void* arg), void* stacktop, int flags, void* arg);
```

### Return Value

`process ID` If negative, `errno` has the exact error code.

### Parameters

`fn` The home for the child task. Create a function (or procedure) that accepts a `void*` parameter argument. When the routine attempts to return, the operating system terminates the task for you.

`stacktop` You must create a stack for the child task. This parameter points to the top of that stack (the highest address of the data block). Because you provide the stack, the stack is fixed in size and cannot grow like a normal task’s stacks.

`flags` Two types of information arithmetically ORed together; the VM spaces to share and the termination signal. This flag supports all signal types and, when the task terminates, the operating system raises the signal you define.

The available VM spaces are as follows:

- `CLONE_VM` Share the data space between tasks. Use this flag to share all static data, preinitialized data, and the allocation heap. Otherwise, copy data space.
- `CLONE_FS` Share the file system information: current working directory, root file system, and default file creation permissions. Otherwise, copy settings.
- `CLONE_FILES` Share open files. When one task changes the file pointer, the other tasks see the change. Likewise, if the task closes the file, the other tasks are not able to access the file any longer. Otherwise, create new references to open inodes.

- `CLONE_SIGHAND` Share signal tables. Individual tasks may choose to ignore open signals (using `sigprocmask()`) without affecting peers. Otherwise, copy tables.
- `CLONE_PID` Share Process ID. Use this flag carefully; not all the existing tools support this feature. The PThreads library does not use this option. Otherwise, allocate new PID.

`arg` You can pass a pointer reference to any data value using this parameter. When the operating system finishes creating the child task, it calls the routine `fn` with the `arg` parameter. If you use this feature, be sure to place the value `arg` points to in the shared data region (`CLONE_VM`).

### Possible Errors

`EAGAIN` The `__clone()` cannot allocate sufficient memory to copy the parent's page tables and allocate a task structure for the child.

`ENOMEM` The `__clone()` failed to allocate the necessary kernel structures because memory is tight.

### Example

```
#define STACKSIZE   1024

void Child(void* arg)
{
    /*---child's responsibility---*/
    exit(0);
}

...
int main(void)
{   int cchild;
    char *stack=malloc(STACKSIZE);

    if ( (cchild = __clone(&Child, stack+STACKSIZE-1,
          SIGCHLD,  0) == 0 )
```

## exec()

Run an external program (either a binary or an executable script with `#! <interpreter> [arg]` in the first line). This call replaces the currently running task with the external program's context. The new program keeps the caller's PID and open files.

The calls `execl()`, `execlp()`, `execle()`, `execv()`, and `execvp()` are all front ends to `execve()`.

## Prototype

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

## Return Value

This call does not return if successful. If it fails, the return value is `-1`.

## Parameters

`file` The program to execute. The call searches for the name in this variable using the defined `PATH`.

`path` The absolute path and filename of the program to execute.

`argv` The string array of command-line parameters. The first array element value must be `arg0` (or the name of the program). The last array element is always zero (`0`).

`arg` A command-line parameter. This is followed by an ellipsis (…) to indicate that there are several arguments. The first `arg` is always the name of the program, and the last `arg` is always zero (`0`).

`envp` The string array of environment parameters. Each parameter is in the form `<`*`param`*`>=<`*`value`*`>` (for example, `TERM=vt100`). The last array element is always zero (`0`).

## Possible Errors

`EACCES` The file or a script interpreter is not a regular file, or execute permission is denied for the file or a script interpreter, or the file system is mounted `noexec`.

`EPERM` The file system is mounted `nosuid`, the user is not the superuser, and the file has an SUID or SGID bit set.

`EPERM` The process is being traced, the user is not the superuser, and the file has an SUID or SGID bit set.

`E2BIG` The argument list is too big.

`ENOEXEC` An executable is not in a recognized format, is for the wrong architecture, or has some other format error that means it cannot be executed.

`EFAULT` The filename points outside your accessible address space.

`ENAMETOOLONG` The filename is too long.

| | |
|---|---|
| ENOENT | The filename or a script or ELF interpreter does not exist. |
| ENOMEM | Insufficient kernel memory was available. |
| ENOTDIR | A component of the path prefix of filename, script, or ELF interpreter is not a directory. |
| EACCES | Search permission is denied on a component of the path prefix of filename or the name of a script interpreter. |
| ELOOP | Too many symbolic links were encountered in resolving filename, the name of a script, or ELF interpreter. |
| ETXTBUSY | Executable was open for writing by one or more processes. |
| EIO | An I/O error occurred. |
| ENFILE | The limit on the total number of files open on the system has been reached. |
| EMFILE | The process has the maximum number of files open. |
| EINVAL | An ELF executable had more than one PT_INTERP segment. |
| EISDIR | An ELF interpreter was a directory. |
| ELIBBAD | An ELF interpreter was not in a recognized format. |

### Example

```
execl("/bin/ls", "/bin/ls", "-al", "/home", "/boot", 0);
perror("execl() failed"); /* No IF needed here: if successful, no return */

char *args[]={"ls", "-al", "/home", "/boot", 0};
execvp(args[0], args);
perror("execvp() failed");
```

## sched_yield()

Relinquish control of the CPU without blocking. This routine tells the scheduler that the currently running task wants to give up the remains of its current timeslice. The call returns on the next timeslice.

### Prototype

```
#include <sched.h>
int sched_yield(void);
```

### Return Value

Zero if all goes okay and control is transferred; otherwise, -1.

### Parameters

(none)

### Possible Errors

(none defined)

### Example

```
#include <sched.h>
sched_yield();
```

## wait(), waitPID()

Wait for and acknowledge the termination of a child process. This is important to keep zombie processes from lingering in the process table and to free up valuable resources. The `wait()` call waits for any process to terminate, and the `waitPID()` call permits you to specify a specific process or group. You can use the following macros to get the meaning from the status:

- `WIFEXITED(status)` is non-zero if the child exited normally.
- `WEXITSTATUS(status)` evaluates to the least significant eight bits of the return code of the child that terminated, which may have been set as the argument to a call to `exit()` or as the argument for a return statement in the main program. This macro can only be evaluated if `WIFEXITED` returned non-zero.
- `WIFSIGNALED(status)` returns `true` if the child process exited because of a signal that was not caught.
- `WTERMSIG(status)` returns the number of the signal that caused the child process to terminate. This macro can only be evaluated if `WIFSIGNALED` returned non-zero.
- `WIFSTOPPED(status)` returns `true` if the child process that caused the return is currently stopped; this is only possible if the call was done using `WUNTRACED`.
- `WSTOPSIG(status)` returns the number of the signal that caused the child to stop. This macro can only be evaluated if `WIFSTOPPED` returned non-zero.

### Prototype

```
#include <sys/types.h>
#include <sys/wait.h>
PID_t wait(int *status);
PID_t waitpid(PID_t PID, int *status, int options);
```

### Return Value

Both calls return the PID of the child that terminated.

### Parameters

`status` Returns the ending status of the child. If not zero or `NULL`, this parameter picks up the child's termination code and `exit()` value.

`PID` Indicates which process to wait for:

`< -1` Wait for any child process whose process group ID is equal to the absolute value of PID.

`== -1` Wait for any child process; this is the same behavior that `wait()` exhibits.

`== 0` Wait for any child process whose process group ID is equal to that of the calling process.

`> 0` Wait for the child whose process ID is equal to the value of PID.

`options` — `WNOHANG` Return immediately if no child has exited.

`WUNTRACED` Return for children who are stopped and whose status has not been reported.

### Possible Errors

`ECHILD` — If the process specified in PID does not exist or is not a child of the calling process. (This can happen for one's own child if the action for `SIGCHLD` is set to `SIG_IGN`.)

`EINVAL` — If the options argument was invalid.

`EINTR` — If `WNOHANG` was not set and an unblocked signal or a `SIGCHLD` was caught. Just try again.

### Example

```
void sig_child(int signum) /* This handler only gets one waiting zombie */
{   int status;
    wait(&status);
    if ( WIFEXITED(status) )
        printf("Child exited with the value of %d\n", WEXITSTATUS(status));
    if ( WIFSIGNALED(status) )
        printf("Child aborted due to signal #%d\n", WTERMSIG(status));
    if ( WIFSTOPPED(status) )
        printf("Child stopped on signal #%d\n", WSTOPSIG(signal));
}

void sig_child(int signum)  /* This handler removes all waiting zombies */
{
    while ( waitpid(-1, 0, WNOHANG) > 0 );
}
```

# Threads

Threads are another kind of task. This section defines a few library calls from the `pThreads` library.

### `pthread_create()`

This call creates a lightweight kernel process (thread). The thread starts in the function that `start_fn` points to using `arg` as the function's parameter. When the function returns, the thread terminates. The function should return a `void*` value, but if it doesn't, the thread still terminates and the result is set to `NULL`.

### Prototype

```
#include <pthread.h>
int pthread_create(pthread_t *tchild, pthread_attr_t *attr,
void (*start_fn)(void *), void *arg);
```

### Return Value

This is a positive value if successful. If the thread-create call encountered any errors, the call returns a negative value and sets `errno` to the error.

### Parameters

`thread` The thread handle (passed by reference). If successful, the call places the thread handle in this parameter.

`attr` The thread's starting attributes. See `pthread_attr_init` for more information.

`start_fn` The routine in which the thread is to start. This function should return a `void*` value.

`arg` The parameter passed to `start_fn`. You should make this parameter a nonshared (unless you plan on locking it), nonstack memory reference.

### Possible Errors

`EAGAIN` Not enough system resources to create a process for the new thread.

`EAGAIN` More than `PTHREAD_THREADS_MAX` threads are already active.

### Example

```
void* child(void *arg)
{
    /**** Do something! ****/
    pthread_exit(arg); /* terminate and return arg */
}

int main()
{   pthread_t tchild;

    if ( pthread_create(&tchild, 0, child, 0) < 0 )
        perror("Can't create thread!");
    /**** Do something! ****/
    if ( pthread_join(tchild, 0) != 0 )
        perror("Join failed");
}
```

## pthread_join()

Similar to the wait() system call, this call waits for and accepts the return value of the child thread.

### Prototype

```
#include <pthread.h>
int pthread_join(pthread_t tchild, void **retval);
```

### Return Value

A positive value if successful. If the thread-create call encountered any errors, the call returns a negative value and sets errno to the error.

### Parameters

| | |
|---|---|
| thread | The thread handle to wait on |
| retval | The pointer to the value passed back (passed by reference) |

### Possible Errors

| | |
|---|---|
| ESRCH | No thread could be found corresponding to that specified by tchild. |
| EINVAL | The tchild thread has been detached. |
| EINVAL | Another thread is already waiting on termination of tchild. |
| EDEADLK | The tchild argument refers to the calling thread. |

### Example

(see pthread_create())

## pthread_exit()

Explicitly terminates the current thread, returning retval. You can use a simple return statement as well.

### Prototype

```
#include <pthread.h>
void pthread_exit(void *retval);
```

### Return Value

(none)

### Parameter

| | |
|---|---|
| retval | The void* value to return. Make sure that this value is non-stack memory. |

### Possible Errors

(none)

### Example

(see pthread_create())

## pthread_detach()

Detaches tchild thread from the parent. Normally, you need to join or wait for every process and thread. This call lets you create several threads and ignore them. This is the same as setting the thread's attribute upon creation.

### Prototype

```
#include <pthread.h>
int pthread_detach(thread_t tchild);
```

### Return Value

A zero if successful. If the thread-create call encountered any errors, the call returns a negative value and sets errno to the error.

### Parameter

| | |
|---|---|
| tchild | The child thread to detach |

### Possible Errors

| | |
|---|---|
| ESRCH | No thread could be found corresponding to that specified by tchild. |
| EINVAL | The tchild thread has been detached. |
| EINVAL | Another thread is already waiting on termination of tchild. |
| EDEADLK | The tchild argument refers to the calling thread. |

### Example

```
void* child(void *arg)
{
    /**** Do something! ****/
    pthread_exit(arg); /* terminate and return arg */
}

int main()
{   pthread_t tchild;

    if ( pthread_create(&tchild, 0, child, 0) < 0 )
        perror("Can't create thread!");
    else
        pthread_detach(tchild);
    /**** Do something! ****/
}
```

# Locking

The primary advantage of using threads is sharing data memory. Because the threads may try to revise the memory at the same time, you need to lock the memory for exclusive access. This section describes `pThread` calls that you can use (even with clones) to lock memory.

## pthread_mutex_init(), pthread_mutex_destroy()

These calls create and destroy mutex semaphore variables. You may not need the initializer because the defined variables are easier and faster to use. The `destroy` call normally frees up any resources. However, the Linux implementation uses no allocated resources, so the call does nothing more than check whether the resource is unlocked.

### Prototype

```
#include <pthread.h>

/*---Predefined mutex settings---*/
pthread_mutex_t fastmutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t recmutex = PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;
pthread_mutex_t errchkmutex = PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP;

int pthread_mutex_init(pthread_mutex_t *mutex,
           const pthread_mutexattr_t *mutexattr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

### Return Value

Always zero.

### Parameters

| | |
|---|---|
| `mutex` | The mutex to create or destroy. |
| `mutexattr` | Any attributes to set. If `NULL`, the call uses the default setting (`PTHREAD_MUTEX_INITIALIZER`). |

### Possible Errors

(none)

## pthread_mutex_lock(), pthread_mutex_trylock()

Lock or try to lock a semaphore for entering a critical section. The parameter is simply a variable that acts like a reservation ticket. If another thread tries to lock a reserved spot, it blocks until the reserving thread releases the semaphore.

### Prototype

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

### Return Value

The call returns zero on success and nonzero on error. You can find the exact code in `errno`.

### Parameter

| | |
|---|---|
| `mutex` | The semaphore variable |

### Possible Errors

| | |
|---|---|
| `EINVAL` | The mutex has not been properly initialized. |
| `EDEADLK` | (`pthread_mutex_try_lock`) The calling thread has already locked the mutex (error-checking mutexes only). |
| `EBUSY` | (`pthread_mutex_lock`) The calling thread can't acquire because it is currently locked. |

### Example

```
pthread_mutex_t mutex = fastmutex;
...
if ( pthread_mutex_lock(&mutex) == 0 )
{
    /**** work on critical data ****/
    pthread_mutex_unlock(&mutex);
}
```

```
pthread_mutex_t mutex = fastmutex;
...
/*---Do other processing while waiting for semaphore---*/
while ( pthread_mutex_trylock(&mutex) != 0  &&  errno == EBUSY )
{
    /**** Work on something else while waiting ****/
}
/*---Got the semaphore!  Now work on the critical section---*/
if ( errno != ENOERROR )
{
    /**** work on critical data ****/
    pthread_mutex_unlock(&mutex);
}
```

## `pthread_mutex_unlock()`

Unlock a mutex semaphore.

### Prototype

```
#include <pthread.h>
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

### Return Value

The call returns zero on success and nonzero on error. You can find the exact code in `errno`.

### Parameter

| | |
|---|---|
| `mutex` | The semaphore variable |

### Possible Errors

| | |
|---|---|
| `EINVAL` | The mutex has not been properly initialized. |
| `EPERM` | The calling thread does not own the mutex (error-checking mutexes only). |

### Example

(see `pthread_mutex_lock()`)

# Signals

When working with tasks, your program may get signals (or asynchronous notifications). This section describes system calls that let you capture and process them.

## `signal()`

Register the `sig_fn` routine to answer the `signum` signal. The default behavior is a single shot; the signal handler reverts to the default after getting the first signal. Use `sigaction()` instead if you want to control the behavior more.

### Prototype

```
#include <signal.h>
void (*signal(int signum, void (*sig_fn)(int signum)))(int signum);
-or-
typedef void (*TSigFn)(int signum);
TSigFn signal(int signum, TSigFn sig_fn);
```

### Return Value

A positive value if successful. If the thread-create call encountered any errors, the call returns a negative value and sets `errno` to the error.

### Parameters

| | |
|---|---|
| `signum` | The signal number to capture |
| `sig_fn` | The program routine that the schedule calls |

### Possible Error

(`errno` not set)

### Example

```
void sig_handler(int signum)
{
    switch ( signum )
    {
        case SIGFPE:
...
    }
}

...
if ( signal(SIGFPE, sig_handler) == 0 )
    perror("signal() failed");
```

## sigaction()

Similar to `signal()`, `sigaction()` establishes the receiver of certain signals. Unlike `signal()`, however, this call gives you a lot more control over how the signaling notification behaves. It is also a little more complicated to use.

### Prototype

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *sigact,
        struct sigaction *oldsigact);
```

### Return Value

Zero upon success; otherwise, nonzero.

### Parameters

`signum` The signal to capture.

`sigact` The desired behavior and signal handler, using the following structure:

```
struct sigaction
{
     void (*sa_handler)(int);
     sigset_t sa_mask;
     int sa_flags;
     void (*sa_restorer)(void);
};
```

`sa_handler` Signal handler function pointer.

`sa_mask` The set of signals to block while servicing a signal in the signal handler.

`sa_restorer` Obsolete; do not use.

`sa_flags` How to handle the signals. You can use the following flags:

- `SA_NOCLDSTOP` If the signal is `SIGCHLD`, ignore cases when the child stops or pauses.
- `SA_ONESHOT` or `SA_RESETHAND` Reset the handler to the default after getting the first signal.
- `SA_RESTART` Try to restart an interrupted system call. Normally, system calls that are interrupted return an `EINTR` error. This option tries to restart the call and avoid `EINTR` errors.
- `SA_NOMASK` or `SA_NODEFER` Allow like signals to interrupt the handler. Normally, if your handler is responding to a particular signal like `SIGCHLD`, the kernel suspends other `SIGCHLD` signals. This can lead to lost signals. Using this option permits your handler to be interrupted. Be careful using this option.

`oldsigact` A repository of the old behaviors. You can copy the old settings here.

## Possible Errors

`EINVAL` An invalid signal was specified. This will also be generated if an attempt is made to change the action for `SIGKILL` or `SIGSTOP` that cannot be caught.

`EFAULT` The `sigact` or `oldsigact` parameter points to memory that is not a valid part of the process address space.

`EINTR` System call was interrupted.

## Example

```
void sig_handler(int signum)
{
    switch ( signum )
    {
        case SIGCHLD:
...
    }
}

...
struct sigaction sigact;
bzero(&sigact, sizeof(sigact));
sigact.sa_handler = sig_handler;  /* set the handler */
sigact.sa_flags = SA_NOCLDSTOP | SA_RESTART; /* set options */
if ( sigaction(SIGCHLD, &sigact, 0) == 0 )
    perror("sigaction() failed");
```

### `sigprocmask()`

Sets which signals are permitted to interrupt while servicing a signal.

#### Prototype

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *sigset, sigset_t *oldsigset);
```

#### Return Value

Nonzero upon error; otherwise, zero.

#### Parameters

`how` The following are how the interrupting signals are treated while servicing a signal:

- `SIG_BLOCK` The set of blocked signals is the union of the current set and the `sigset` argument.
- `SIG_UNBLOCK` The signals in `sigset` are removed from the current set of blocked signals. It is legal to attempt to unblock a signal that is not blocked.
- `SIG_SETMASK` The set of blocked signals is set to the argument `sigset`.

`sigset` The destination signal-set.

`Oldsigset` If non-`NULL`, the call places a copy of the old values in here.

#### Possible Errors

`EFAULT` The `sigset` or `oldsigset` parameter points to memory that is not a valid part of the process address space.

`EINTR` System call was interrupted.

## Files and So On

This section describes a few library and system calls for file management.

### `bzero()`, `memset()`

`bzero()` initializes the specified block to zeros. This call is deprecated, so you might want to use `memset()` instead.

`memset()` sets the specified block to `val`.

#### Prototype

```
#include <string.h>
void bzero(void *mem, int bytes);
void* memset(void *mem, int val, size_t bytes);
```

### Return Value

`bzero()` returns no value.

`memset()` returns the reference `mem`.

### Parameters

| | |
|---|---|
| `mem` | The memory segment to initialize |
| `val` | The value to fill the segment with |
| `bytes` | The number of bytes to write (the size of the memory segment) |

### Possible Errors

(none)

### Example

```
bzero(&addr, sizeof(addr));
```

```
memset(&addr, 0, sizeof(addr));
```

## `fcntl()`

Manipulate the file or socket handle.

### Prototype

```
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
int fcntl(int fd, int cmd, struct flock *flock);
```

### Return Value

On error, `-1` is returned and `errno` is set appropriately. For a successful call, the return value depends on the operation:

| | |
|---|---|
| `F_DUPFD` | The new descriptor |
| `F_GETFD` | Value of flag |
| `F_GETFL` | Value of flags |
| `F_GETOWN` | Value of descriptor owner |
| `F_GETSIG` | Value of signal sent when read or write becomes possible, or zero for traditional `SIGIO` behavior |

All other commands return zero.

### Parameters

`fd` The descriptor to manipulate.

`cmd` The operation to perform. Some operations are duplicates of existing functions. Some operations require an operand (`arg` or `flock`). Each operation is grouped into specific functions:

- *Duplicate descriptor* (`F_DUPFD`) Same as `dup2(arg, fd)`, this operation replaces `fd` with a copy of the descriptor in `arg`.
- *Manipulate close-on-exec* (`F_GETFD`, `F_SETFD`) The kernel does not pass all file descriptors to the exec-child process. With this parameter, you can test or set the close-on-exec.
- *Manipulate descriptor flags* (`F_GETFL`, `F_SETFL`) Using these commands, you can get the flags (set by the `open()` system call) of the descriptor. You can only set `O_APPEND`, `O_NONBLOCK`, and `O_ASYNC`.
- *Manipulate file locks* (`F_GETLK`, `F_SETLK`, `F_SETLKW`) `GETLK` retrieves the lock structure that currently holds the file. If the file is not locked
    - *Determine who owns I/O signals* (`F_GETOWN`, `F_SETOWN`)—Return or set the PID of the current owner of the SIGIO signal.
    - *Determine the kind of signal to send* (`F_GETSIG`, `F_SETSIG`)—Gets or sets the signal type when more I/O operations can be performed. Default is `SIGIO`.

`arg` The value to set.

`flock` The locking key.

### Possible errors

`EACCES` Operation is prohibited by locks held by other processes.

`EAGAIN` Operation is prohibited because the file has been memory-mapped by another process.

`EBADF` `fd` is not an open file descriptor.

`EDEADLK` It was detected that the specified `F_SETLKW` command would cause a deadlock.

`EFAULT` `lock` is outside your accessible address space.

| | |
|---|---|
| EINTR | For F_SETLKW, the command was interrupted by a signal. For F_GETLK and F_SETLK, the command was interrupted by a signal before the lock was checked or acquired—most likely when locking a remote file (locking over NFS), but it can sometimes happen locally. |
| EINVAL | For F_DUPFD, arg is negative or is greater than the maximum allowable value. For F_SETSIG, arg is not an allowable signal number. |
| EMFILE | For F_DUPFD, the process already has the maximum number of file descriptors open. |
| ENOLCK | Too many segment locks open, lock table is full, or a remote locking protocol failed (locking over NFS, for example). |
| EPERM | Attempted to clear the O_APPEND flag on a file that has the append-only attribute set. |

### Example

```
#include <unistd.h>
#include <fnctl.h>
...
printf("PID which owns SIGIO: %d",
    fnctl(fd, F_GETOWN));
```

```
#include <unistd.h>
#include <fnctl.h>
...
if ( fnctl(fd, F_SETSIG, SIGKILL) != 0 )
    perror("Can't set signal");
```

```
#include <unistd.h>
#include <fnctl.h>
...
if ( (fd_copy = fcntl(fd, F_DUPFD)) < 0 )
    perror("Can't dup fd");
```

## pipe()

Creates a pipe that points to itself. Each file descriptor in fd[] coincides with input (fd[0]) and output (fd[1]). If you write to fd[1], you can read the data on fd[0]. Used mostly with fork().

### Prototype

```
#include <unistd.h>
int pipe(fd[2]);
```

### Return Value

Zero if okay; -1 on error.

### Parameter

`fd` An array of two integers to receive the new file descriptor values

### Possible Errors

`EMFILE` Too many file descriptors are already in use by the current process.

`ENFILE` The system's file table is full.

`EFAULT` The process does not own the memory that `fd` points to (invalid memory reference).

### Example

```
int fd[2];
pipe(fd); /* create pipe */
```

## poll()

Similar to `select()`, this call waits on any one of several I/O channels for changes. Instead of using macros for managing and controlling the descriptor list, the programmer uses structure entries.

### Prototype

```
#include <sys/poll.h>
int poll(struct pollfd *ufds, unsigned int nfds, int timeout);
```

### Return Value

If less than zero, an error occurred; a zero returned means that the call timed out. Otherwise, the call returns the number of descriptor records that changed.

### Parameters

`ufds` The following is an array of `pollfd` structures. Each record tracks a different file descriptor.

```
struct pollfd
{
    int fd;            /* file descriptor */
    short events;      /* requested events */
    short revents;     /* returned events */
};
```

The `fd` field is the file descriptor to check. The events and `revents` fields indicate the events to check and the events that occurred, respectively. The bit-values available are as follows:

`POLLIN` There is data to read.

`POLLPRI` There is urgent data to read.

`POLLOUT` Writing now will not block.

`POLLERR` Error condition.

`POLLHUP` Hung up.

`POLLNVAL` Invalid request; `fd` not open.

`POLLRDNORM` Normal read (Linux only).

`POLLRDBAND` Read out-of-band (Linux only).

`POLLWRNORM` Normal write (Linux only).

`POLLWRBAND` Write out-of-band (Linux only).

`nfds` The number of records to check during the call.

`timeout` The timeout in milliseconds. If `timeout` is negative, the call waits forever.

## Possible Errors

`ENOMEM` There was no space to allocate file descriptor tables.

`EFAULT` The array given as argument was not contained in the calling program's address space.

`EINTR` A signal occurred before any requested event.

## Example

```
int fd_count=0;
struct pollfd fds[MAXFDs];
fds[fd_count].fd = socket(PF_INET, SOCK_STREAM, 0);
/*** bind() and listen() socket ***/
fds[fd_count++].events = POLLIN;
for (;;)
{
    if ( poll(fds, fd_count, TIMEOUT_MS) > 0 )
    {   int i;
        if ( (fds[0].revents & POLLIN) != 0 )
        {
            fds[fd_count].events = POLLIN | POLLHUP;
            fds[fd_count++].fd = accept(fds[0].fd, 0, 0);
        }
        for ( i = 1; i < fd_count; i++ )
        {
            if ( (fds[i].revents & POLLHUP) != 0 )
            {
                close(fds[i].fd);
                /*** Move up FDs to fill empty slot ***/
                fd_count--;
            }
            else if ( (fds[i].revents & POLLIN) != 0 )
```

```
                /*** Read and process data ***/
        }
    }
}
```

## `read()`

Read `buf_len` bytes from the `fd` file descriptor into the buffer. You can use this system call for sockets as well as files, but this call does not provide as much control as the `recv()` system call.

### Prototype

```
#include <unistd.h>
int read(int fd, char *buffer, size_t buf_len);
```

### Return Value

The number of bytes actually read.

### Parameters

| | |
|---|---|
| `fd` | File (or socket) descriptor |
| `buffer` | The memory buffer to accept the read data |
| `buf_len` | The number of bytes to read and the number of legal bytes in the buffer |

### Possible Errors

| | |
|---|---|
| `EINTR` | The call was interrupted by a signal before any data was read. |
| `EAGAIN` | Non-blocking I/O has been selected using `O_NONBLOCK` and no data was immediately available for reading. |
| `EIO` | I/O error. This will happen when the process is in a background process group, tries to read from its controlling tty, is either ignoring or blocking `SIGTTIN`, or its process group is orphaned. It can also occur when there is a low-level I/O error while reading from a disk or tape. |
| `EISDIR` | `fd` refers to a directory. |
| `EBADF` | `fd` is not a valid file descriptor or is not open for reading. |
| `EINVAL` | `fd` is attached to an object that is unsuitable for reading. |
| `EFAULT` | `buf` is outside your accessible address space. |

### Example

```
int sockfd;
int bytes_read;
char buffer[1024];
/*---create socket & connect to server---*/
if ( (bytes_read = read(sockfd, buffer, sizeof(buffer))) < 0 )
    perror("read");
```

### select()

Wait for any I/O status changes from the file descriptor sets. When any of the specified sets changes, the call returns. You have four macros to help construct and manage the file descriptor sets:

- FD_CLR Remove a descriptor from the set.
- FD_SET Add a descriptor to a set.
- FD_ISSET Test if specified descriptor is ready for I/O.
- FD_ZERO Initialize the set to empty.

#### Prototype

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
int select(int hi_fd, fd_set *readfds, fd_set *writefds,
fd_set *exceptfds, struct timeval *timeout);
FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
FD_SET(int fd, fd_set *set);
FD_ZERO(fd_set *set);
```

#### Return Value

The number of descriptors that have changed states. If an error occurred, the return value is negative. If the timeout expired, the return value is zero.

#### Parameters

| | |
|---|---|
| hi_fd | This is the highest file descriptor number + 1. For example, if you have four files open plus the stdio, your descriptors could be 0, 1, 2, 3, 5, 6, and 8. The highest is 8. If you include fd(8) in your select statement, hi_fd would equal 9. If the highest fd were 5, this parameter would be 6. |
| readfds | The set of descriptors to test for readability. |
| writefds | The set of descriptors to test for writing. |
| exceptfds | The set of descriptors to test for out-of-band data. |
| timeout | The maximum time to wait for data to arrive in microseconds. This is a pointer to a number. If the number is zero (not the pointer), the call returns immediately after checking all the descriptors. If the pointer is NULL (zero), the select's timeout feature is disabled. |
| fd | The file descriptor to add, remove, or test. |
| set | The file descriptor set. |

### Possible Errors

| | |
|---|---|
| EBADF | An invalid file descriptor was given in one of the sets. |
| EINTR | A non-blocked signal was caught. |
| EINVAL | n is negative. |
| ENOMEM | select was unable to allocate memory for internal tables. |

### Example

```
int i, ports[]={9001, 9002, 9004, -1};
int sockfd, max=0;
fd_set set;
struct sockaddr_in addr;
struct timeval timeout={2,500000}; /* 2.5 sec. */

FD_ZERO(&set);
bzero(&addr, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = INADDR_ANY;
for ( i = 0; ports[i] > 0; i++ )
{
    sockfd = socket(PF_INET, SOCK_STREAM, 0);
    addr.sin_port = htons(ports[i]);
    if ( bind(sockfd, &addr, sizeof(addr)) != 0)
        perror("bind() failed");
    else
    {
        FD_SET(sockfd, &set);
        if ( max < sockfd )
            max = sockfd;
    }
}
if ( select(max+1, &set, 0, &set, &timeout) > 0 )
{
    for ( i = 0; i <= max; i++ )
        if ( FD_ISSET(i, &set) )
        {   int client = accept(i, 0, 0);
            /**** process the client's requests ****/
        }
}
```

## write()

Write msg_len bytes to fd field descriptor from buffer. You can use a socket descriptor as well, but it does not provide you with as much control as the send() system call.

### Prototype

```
#include <unistd.h>
int write(int fd, const void *buffer, size_t msg_len);
```

### Return Value

Number of bytes written. The byte count can be less than `msg_len`. If the call does not succeed in writing all required bytes, you can use a loop for successive writes. If negative, the call stores the error detail in `errno`.

### Parameters

| | |
|---|---|
| `fd` | File descriptor (can be a socket descriptor) |
| `buffer` | The message to write |
| `msg_len` | The length of the message |

### Possible Errors

| | |
|---|---|
| `EBADF` | `fd` is not a valid file descriptor or is not open for writing. |
| `EINVAL` | `fd` is attached to an object that is unsuitable for writing. |
| `EFAULT` | `buf` is outside your accessible address space. |
| `EPIPE` | `fd` is connected to a pipe or socket whose reading end is closed. When this happens, the writing process will receive a `SIGPIPE` signal; if it catches, blocks, or ignores the error, `EPIPE` is returned. |
| `EAGAIN` | Non-blocking I/O has been selected using `O_NONBLOCK` and there was no room in the pipe or socket connected to `fd` to write the data immediately. |
| `EINTR` | The call was interrupted by a signal before any data was written. |
| `ENOSPC` | The device containing the file referred to by `fd` has no room for the data. |
| `EIO` | A low-level I/O error occurred while modifying the inode. |

### Example

```
/*** Write a message (TCP, UDP or Raw) ***/
int sockfd;
int bytes, bytes_wrote=0;
/*--- Create socket, connect to server ---*/
while ( (bytes = write(sockfd, buffer, msg_len)) > 0 )
    if ( (bytes_wrote += bytes) >= msg_len )
        break;
if ( bytes < 0 )
    perror("write");
```

## close()

Closes all descriptors (file or socket). If the socket is connected to a server or client, it requests a close(). The channel actually remains active after the close until the channel empties or times out. Every process has a limit to the number of open descriptors it can have. getdtablesize() returns 1024 in Linux 2.2.14, and the /usr/include/linux/limits.h file defines this limit with NR_OPEN. Also, the first three descriptors default to stdin (0), stdout (1), and stderr (2).

### Prototype

```
#include <unistd.h>
int close(int fd);
```

### Return Value

Zero if everything goes well. If an error occurs, you can find the cause in errno.

### Parameter

| | |
|---|---|
| fd | The file or socket descriptor |

### Possible Error

| | |
|---|---|
| EBADF | fd isn't a valid open file descriptor. |

### Example

```
int sockfd;
sockfd = socket(PF_INET, SOCK_RAW, htons(99));
if ( sockfd < 0 )
    PANIC("Raw socket create failed");
...
if ( close(sockfd) != 0 )
    PANIC("Raw socket close failed");
```