Practical Case Study B

Operating Systems Programming – 300698

1 Introduction

A Shell or Command Language Interpreter (CLI) is a piece of software that provides a simple, but powerful, textual interface for users to an operating system. In this Case Study you will investigate the operation of and implement a simple CLI.

2 Specification

A CLI accepts textual input from the user, and executes the commands issued. The main logic of the CLI is given below:

```
main:
    loop
        get input line
        if end of input exit
        break line into words
        found := false
        if command is builtin
        then
          do_builtin(line)
          found:=true
        else
          found:=find_and_execute(line)
        end if
        if not found report error
      end loop
```

Your first task will be to write a program that repeatedly reads a line of input from the user, the `fgets()` function (see Sect. 5 for usage information of all system provided functions) will help you here. Your program should end when either end of the input file is encountered, or the exact word `exit` appears in the input as the only word on a line. Traditionally both the File System and Command Language are case sensitive, you should also implement this.

Your next task will be to break the line up into words, which are separated by one or more spaces. The provided function `tokenize()` does this, and it uses `strtok()` internally. You may use this function if you wish, in which case you should provide documentation on its operation, or you may write your own parser.

You should next implement the `find_and_execute()` section of the logic above, by creating a new process using `fork()`, and then use one of the `exec()` family of functions to run the program requested by the user in the text provided. If the requested program cannot be run then an appropriate error message should be displayed, `perror()` will help with this (as there are many reasons why this may fail), and the child process terminated.

The CLI process must pause until the created process is concluded, `wait()` will need to be used here. Once the new process has finished you must decode and print out the exit status of that process.

Once this works you should add a builtin function `cd` to change the working directory of the CLI. This builtin should always use the next word supplied as the directory to change to, failure to supply a destination should be treated as an error. The `chdir()` function will be vital here.

Note This logic has an infinite loop in it. The appropriate place to determine when to exit is in the middle of the loop. When testing the sample code (if used) you should use the Ctrl + C key combination to break out of the program. Do not use Ctrl + Z, it does something completely different.

3 Marking Scheme

Please see the rubric on the vUWS site for the marking Scheme.

4 Sample Code

Please note that in the sample code "␣" indicates a space character in a string.

```
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
#include<sys/types.h>
#include<sys/wait.h>
#include<unistd.h>
#include<errno.h>

#define MAX_LINE 4096
#define MAX_WORDS MAX_LINE/2
/* a line can have at most MAX_LINE/2 words, why? */

void tokenize(char *line, char **words, int *nwords);
/* break line into words separated by whitespace, placing them in the
   array words, and setting the count to nwords */

int main()
{
        char line[MAX_LINE], *words[MAX_WORDS], message[MAX_LINE];
        int stop=0,nwords=0;

        while(1)
        {
                printf("OSP␣CLI␣$␣");

                /* read a line of text here */

                tokenize(line,words,&nwords);

                /* More to do here */

        }
        return 0;
}

/* this function works, it is up to you to work out why! */
void tokenize(char *line, char **words, int *nwords)
{
        *nwords=1;

        for(words[0]=strtok(line,"␣\t\n");
            (*nwords<MAX_WORDS)&&(words[*nwords]=strtok(NULL, "␣\t\n"));
            *nwords=*nwords+1
           ); /* empty body */
        return;
}
```

5 Supplementary Materials

The material on the following pages is an extract of the linux system documentation and may prove useful in implementing this Workshop. These manual pages are taken from the Linux *man-pages* Project available at:
`http://www.kernel.org/doc/man-pages/`.

NAME

fgetc, fgets, getc, getchar, gets, ungetc − input of characters and strings

SYNOPSIS

#include <stdio.h>

int fgetc(FILE ****stream*);**
char *fgets(char ****s*, int** *size***, FILE ****stream***);**
int getc(FILE ****stream*);**
int getchar(void);
char *gets(char ****s*);**
int ungetc(int *c***, FILE ****stream***);**

DESCRIPTION

fgetc() reads the next character from *stream* and returns it as an *unsigned char* cast to an *int*, or **EOF** on end of file or error.

getc() is equivalent to **fgetc**() except that it may be implemented as a macro which evaluates *stream* more than once.

getchar() is equivalent to **getc(***stdin***)**.

gets() reads a line from *stdin* into the buffer pointed to by *s* until either a terminating newline or **EOF**, which it replaces with **’\0’**. No check for buffer overrun is performed (see **BUGS** below).

fgets() reads in at most one less than *size* characters from *stream* and stores them into the buffer pointed to by *s*. Reading stops after an **EOF** or a newline. If a newline is read, it is stored into the buffer. A **’\0’** is stored after the last character in the buffer.

ungetc() pushes *c* back to *stream*, cast to *unsigned char*, where it is available for subsequent read operations. Pushed-back characters will be returned in reverse order; only one pushback is guaranteed.

Calls to the functions described here can be mixed with each other and with calls to other input functions from the **stdio** library for the same input stream.

For non-locking counterparts, see **unlocked_stdio**(3).

RETURN VALUE

fgetc(), **getc**() and **getchar**() return the character read as an *unsigned char* cast to an *int* or **EOF** on end of file or error.

gets() and **fgets**() return *s* on success, and NULL on error or when end of file occurs while no characters have been read.

ungetc() returns *c* on success, or **EOF** on error.

CONFORMING TO

C89, C99. LSB deprecates **gets**().

BUGS

Never use **gets**(). Because it is impossible to tell without knowing the data in advance how many characters **gets**() will read, and because **gets**() will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security. Use **fgets**() instead.

It is not advisable to mix calls to input functions from the **stdio** library with low-level calls to **read**() for the file descriptor associated with the input stream; the results will be undefined and very probably not what you want.

SEE ALSO

read(2), **write**(2), **ferror**(3), **fgetwc**(3), **fgetws**(3), **fopen**(3), **fread**(3), **fseek**(3), **getline**(3), **getwchar**(3), **puts**(3), **scanf**(3), **ungetwc**(3), **unlocked_stdio**(3)

NAME

strtok, strtok_r − extract tokens from strings

SYNOPSIS

#include <string.h>

char *strtok(char ***_str_, const char ***_delim_**);**

char *strtok_r(char ***_str_, const char ***_delim_**, char ****_saveptr_**);**

DESCRIPTION

The **strtok**() function parses a string into a sequence of tokens. On the first call to **strtok**() the string to be parsed should be specified in *str*. In each subsequent call that should parse the same string, *str* should be NULL.

The *delim* argument specifies a set of characters that delimit the tokens in the parsed string. The caller may specify different strings in *delim* in successive calls that parse the same string.

Each call to **strtok**() returns a pointer to a null-terminated string containing the next token. This string does not include the delimiting character. If no more tokens are found, **strtok**() returns NULL.

A sequence of two or more contiguous delimiter characters in the parsed string is considered to be a single delimiter. Delimiter characters at the start or end of the string are ignored. Put another way: the tokens returned by **strtok**() are always non-empty strings.

The **strtok_r**() function is a reentrant version **strtok**(). The *saveptr* argument is a pointer to a *char ** variable that is used internally by **strtok_r**() in order to maintain context between successive calls that parse the same string.

On the first call to **strtok_r**(), *str* should point to the string to be parsed, and the value of *saveptr* is ignored. In subsequent calls, *str* should be NULL, and *saveptr* should be unchanged since the previous call.

Different strings may be parsed concurrently using sequences of calls to **strtok_r**() that specify different *saveptr* arguments.

EXAMPLE

The following program uses nested loops that employ **strtok_r**() to break a string into a two-level hierarchy of tokens. The first command-line argument specifies the string to be parsed. The second argument specifies the delimiter character(s) to be used to separate that string into "major" tokens. The third argument specifies the delimiter character(s) to be used to separate the "major" tokens into subtokens.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int
main(int argc, char *argv[])
{
   char *str1, *str2, *token, *subtoken;
   char *saveptr1, *saveptr2;
   int j;

   if (argc != 4) {
      fprintf(stderr, "Usage: %s string delim subdelim\n",
            argv[0]);
      exit(EXIT_FAILURE);
```

```
    }

    for (j = 1, str1 = argv[1]; ; j++, str1 = NULL) {
        token = strtok_r(str1, argv[2], &saveptr1);
        if (token == NULL)
            break;
        printf("%d: %s0, j, token);

        for (str2 = token; ; str2 = NULL) {
            subtoken = strtok_r(str2, argv[3], &saveptr2);
            if (subtoken == NULL)
                break;
            printf(" --> %s0, subtoken);
        }
    }

    exit(EXIT_SUCCESS);
} /* main */
```

An example of the output produced by this program is the following:

```
$ ./a.out 'a/bbb///cc;xxx:yyy:' ':;' '/'
1: a/bbb///cc
         --> a
         --> bbb
         --> cc
2: xxx
         --> xxx
3: yyy
         --> yyy
```

BUGS

Avoid using these functions. If you do use them, note that:

These functions modify their first argument.

These functions cannot be used on constant strings.

The identity of the delimiting character is lost.

The **strtok**() function uses a static buffer while parsing, so it's not thread safe. Use **strtok_r**() if this matters to you.

RETURN VALUE

The **strtok**() and **strtok_r**() functions return a pointer to the next token, or NULL if there are no more tokens.

CONFORMING TO

strtok()
SVr4, POSIX.1-2001, 4.3BSD, C89.

strtok_r()
POSIX.1-2001

SEE ALSO

index(3), **memchr**(3), **rindex**(3), **strchr**(3), **strpbrk**(3), **strsep**(3), **strspn**(3), **strstr**(3), **wcstok**(3)

NAME

fork − create a child process

SYNOPSIS

#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);

DESCRIPTION

fork() creates a child process that differs from the parent process only in its PID and PPID, and in the fact that resource utilizations are set to 0. File locks and pending signals are not inherited.

Under Linux, **fork**() is implemented using copy-on-write pages, so the only penalty that it incurs is the time and memory required to duplicate the parent's page tables, and to create a unique task structure for the child.

RETURN VALUE

On success, the PID of the child process is returned in the parent's thread of execution, and a 0 is returned in the child's thread of execution. On failure, a −1 will be returned in the parent's context, no child process will be created, and *errno* will be set appropriately.

ERRORS

EAGAIN
: **fork**() cannot allocate sufficient memory to copy the parent's page tables and allocate a task structure for the child.

EAGAIN
: It was not possible to create a new process because the caller's **RLIMIT_NPROC** resource limit was encountered. To exceed this limit, the process must have either the **CAP_SYS_ADMIN** or the **CAP_SYS_RESOURCE** capability.

ENOMEM
: **fork**() failed to allocate the necessary kernel structures because memory is tight.

CONFORMING TO

SVr4, 4.3BSD, POSIX.1-2001.

EXAMPLE

See **pipe**(2) and **wait**(2).

SEE ALSO

clone(2), **execve**(2), **setrlimit**(2), **unshare**(2), **vfork**(2), **wait**(2), **capabilities**(7)

NAME

execl, execlp, execle, execv, execvp − execute a file

SYNOPSIS

#include <unistd.h>

extern char **environ;

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***[]);**

DESCRIPTION

The **exec**() family of functions replaces the current process image with a new process image. The functions described in this manual page are front-ends for the function **execve**(2). (See the manual page for **execve**() for detailed information about the replacement of the current process.)

The initial argument for these functions is the pathname of a file which is to be executed.

The *const char *arg* and subsequent ellipses in the **execl**(), **execlp**(), and **execle**() functions can be thought of as *arg0*, *arg1*, ..., *argn*. Together they describe a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program. The first argument, by convention, should point to the filename associated with the file being executed. The list of arguments *must* be terminated by a NULL pointer, and, since these are variadic functions, this pointer must be cast **(char *) NULL**.

The **execv**() and **execvp**() functions provide an array of pointers to null-terminated strings that represent the argument list available to the new program. The first argument, by convention, should point to the filename associated with the file being executed. The array of pointers *must* be terminated by a NULL pointer.

The **execle**() function also specifies the environment of the executed process by following the NULL pointer that terminates the list of arguments in the parameter list or the pointer to the argv array with an additional parameter. This additional parameter is an array of pointers to null-terminated strings and *must* be terminated by a NULL pointer. The other functions take the environment for the new process image from the external variable *environ* in the current process.

Some of these functions have special semantics.

The functions **execlp**() and **execvp**() will duplicate the actions of the shell in searching for an executable file if the specified filename does not contain a slash (/) character. The search path is the path specified in the environment by the **PATH** variable. If this variable isn't specified, the default path ‘‘:/bin:/usr/bin’’ is used. In addition, certain errors are treated specially.

If permission is denied for a file (the attempted **execve**() returned **EACCES**), these functions will continue searching the rest of the search path. If no other file is found, however, they will return with the global variable *errno* set to **EACCES**.

If the header of a file isn't recognized (the attempted **execve**() returned **ENOEXEC**), these functions will execute the shell with the path of the file as its first argument. (If this attempt fails, no further searching is done.)

RETURN VALUE

If any of the **exec**() functions returns, an error will have occurred. The return value is −1, and the global variable *errno* will be set to indicate the error.

FILES

/bin/sh

ERRORS

All of these functions may fail and set *errno* for any of the errors specified for the library function **execve**(2).

SEE ALSO

sh(1), **execve**(2), **fork**(2), **ptrace**(2), **fexecve**(3), **environ**(7)

COMPATIBILITY

On some other systems the default path (used when the environment does not contain the variable **PATH**) has the current working directory listed after */bin* and */usr/bin*, as an anti-Trojan-horse measure. Linux uses here the traditional "current directory first" default path.

The behavior of **execlp**() and **execvp**() when errors occur while attempting to execute the file is historic practice, but has not traditionally been documented and is not specified by the POSIX standard. BSD (and possibly other systems) do an automatic sleep and retry if ETXTBSY is encountered. Linux treats it as a hard error and returns immediately.

Traditionally, the functions **execlp**() and **execvp**() ignored all errors except for the ones described above and **ENOMEM** and **E2BIG**, upon which they returned. They now return if any error other than the ones described above occurs.

CONFORMING TO

POSIX.1-2001.

NAME

execve − execute program

SYNOPSIS

#include <unistd.h>

int execve(const char * *filename*, **char *const** *argv***[],**
char *const *envp***[]);**

DESCRIPTION

execve() executes the program pointed to by *filename*. *filename* must be either a binary executable, or a script starting with a line of the form "**#!** *interpreter* [arg]". In the latter case, the interpreter must be a valid pathname for an executable which is not itself a script, which will be invoked as **interpreter** [arg] *filename*.

argv is an array of argument strings passed to the new program. *envp* is an array of strings, conventionally of the form **key=value**, which are passed as environment to the new program. Both *argv* and *envp* must be terminated by a null pointer. The argument vector and environment can be accessed by the called program's main function, when it is defined as **int main(int argc, char *argv[], char *envp[])**.

execve() does not return on success, and the text, data, bss, and stack of the calling process are overwritten by that of the program loaded. The program invoked inherits the calling process's PID, and any open file descriptors that are not set to close-on-exec. Signals pending on the calling process are cleared. Any signals set to be caught by the calling process are reset to their default behaviour. The SIGCHLD signal (when set to SIG_IGN) may or may not be reset to SIG_DFL.

If the current program is being ptraced, a **SIGTRAP** is sent to it after a successful **execve**().

If the set-user-ID bit is set on the program file pointed to by *filename*, and the calling process is not being ptraced, then the effective user ID of the calling process is changed to that of the owner of the program file. i Similarly, when the set-group-ID bit of the program file is set the effective group ID of the calling process is set to the group of the program file.

The effective user ID of the process is copied to the saved set-user-ID; similarly, the effective group ID is copied to the saved set-group-ID. This copying takes place after any effective ID changes that occur because of the set-user-ID and set-group-ID permission bits.

If the executable is an a.out dynamically-linked binary executable containing shared-library stubs, the Linux dynamic linker **ld.so**(8) is called at the start of execution to bring needed shared libraries into memory and link the executable with them.

If the executable is a dynamically-linked ELF executable, the interpreter named in the PT_INTERP segment is used to load the needed shared libraries. This interpreter is typically */lib/ld-linux.so.1* for binaries linked with the Linux libc version 5, or */lib/ld-linux.so.2* for binaries linked with the GNU libc version 2.

RETURN VALUE

On success, **execve**() does not return, on error −1 is returned, and *errno* is set appropriately.

ERRORS

E2BIG The total number of bytes in the environment (*envp*) and argument list (*argv*) is too large.

EACCES
Search permission is denied on a component of the path prefix of *filename* or the name of a script interpreter. (See also **path_resolution**(2).)

EACCES
The file or a script interpreter is not a regular file.

EACCES
Execute permission is denied for the file or a script or ELF interpreter.

EACCES
The file system is mounted *noexec*.

EFAULT
filename points outside your accessible address space.

EINVAL
An ELF executable had more than one PT_INTERP segment (i.e., tried to name more than one interpreter).

EIO An I/O error occurred.

EISDIR
An ELF interpreter was a directory.

ELIBBAD
An ELF interpreter was not in a recognised format.

ELOOP
Too many symbolic links were encountered in resolving *filename* or the name of a script or ELF interpreter.

EMFILE
The process has the maximum number of files open.

ENAMETOOLONG
filename is too long.

ENFILE
The system limit on the total number of open files has been reached.

ENOENT
The file *filename* or a script or ELF interpreter does not exist, or a shared library needed for file or interpreter cannot be found.

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

ENOMEM
Insufficient kernel memory was available.

ENOTDIR
A component of the path prefix of *filename* or a script or ELF interpreter is not a directory.

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.

ETXTBSY
Executable was open for writing by one or more processes.

CONFORMING TO

SVr4, 4.3BSD, POSIX.1-2001. POSIX.1-2001 does not document the #! behavior but is otherwise compatible.

NOTES

SUID and SGID processes can not be **ptrace**()d.

Linux ignores the SUID and SGID bits on scripts.

The result of mounting a filesystem *nosuid* vary between Linux kernel versions: some will refuse execution of SUID/SGID executables when this would give the user powers she did not have already (and return EPERM), some will just ignore the SUID/SGID bits and **exec**() successfully.

A maximum line length of 127 characters is allowed for the first line in a #! executable shell script.

HISTORICAL

With Unix V6 the argument list of an **exec**() call was ended by 0, while the argument list of *main* was ended by −1. Thus, this argument list was not directly usable in a further **exec**() call. Since Unix V7 both are NULL.

SEE ALSO

chmod(2), **fork**(2), **path_resolution**(2), **ptrace**(2), **execl**(3), **fexecve**(3), **environ**(7), **ld.so**(8)

NAME

perror − print a system error message

SYNOPSIS

#include <stdio.h>

void perror(const char **s***);**

#include <errno.h>

const char ****sys_errlist*[];**
int *sys_nerr***;**
int *errno***;**

DESCRIPTION

The routine **perror**() produces a message on the standard error output, describing the last error encountered during a call to a system or library function. First (if *s* is not NULL and **s* is not a null byte ('\0')) the argument string *s* is printed, followed by a colon and a blank. Then the message and a new-line.

To be of most use, the argument string should include the name of the function that incurred the error. The error number is taken from the external variable *errno*, which is set when errors occur but not cleared when non-erroneous calls are made.

The global error list *sys_errlist*[] indexed by *errno* can be used to obtain the error message without the newline. The largest message number provided in the table is *sys_nerr* −1. Be careful when directly accessing this list because new error values may not have been added to *sys_errlist*[].

When a system call fails, it usually returns −1 and sets the variable *errno* to a value describing what went wrong. (These values can be found in <*errno.h*>.) Many library functions do likewise. The function **perror**() serves to translate this error code into human-readable form. Note that *errno* is undefined after a successful library call: this call may well change this variable, even though it succeeds, for example because it internally used some other library function that failed. Thus, if a failing call is not immediately followed by a call to **perror**(), the value of *errno* should be saved.

CONFORMING TO

The function **perror**() and the external *errno* (see **errno**(3)) conform to C89, 4.3BSD, POSIX.1-2001. The externals *sys_nerr* and *sys_errlist* conform to BSD.

NOTE

The externals *sys_nerr* and *sys_errlist* are defined by glibc, but in <*stdio.h*>.

SEE ALSO

err(3), **errno**(3), **error**(3), **strerror**(3)

NAME

errno − number of last error

SYNOPSIS

#include <errno.h>

DESCRIPTION

The <*errno.h*> header file defines the integer variable **errno**, which is set by system calls and some library functions in the event of an error to indicate what went wrong. Its value is significant only when the call returned an error (usually −1), and a function that does succeed is allowed to change **errno**.

Sometimes, when −1 is also a valid successful return value one has to zero **errno** before the call in order to detect possible errors.

errno is defined by the ISO C standard to be a modifiable lvalue of type *int*, and must not be explicitly declared; **errno** may be a macro. **errno** is thread-local; setting it in one thread does not affect its value in any other thread.

Valid error numbers are all non-zero; **errno** is never set to zero by any library function. All the error names specified by POSIX.1 must have distinct values, with the exception of **EAGAIN** and **EWOULDBLOCK**, which may be the same.

Below is a list of the symbolic error names that are defined on Linux. Some of these are marked *POSIX.1*, indicating that the name is defined by POSIX.1-2001, or *C99*, indicating that the name is defined by C99.

E2BIG	Argument list too long (POSIX.1)
EACCES	Permission denied (POSIX.1)
EADDRINUSE	Address already in use (POSIX.1)
EADDRNOTAVAIL	Address not available (POSIX.1)
EAFNOSUPPORT	Address family not supported (POSIX.1)
EAGAIN	Resource temporarily unavailable (may be the same value as **EWOULDBLOCK**) (POSIX.1)
EALREADY	Connection already in progress (POSIX.1)
EBADE	Invalid exchange
EBADF	Bad file descriptor (POSIX.1)
EBADFD	File descriptor in bad state
EBADMSG	Bad message (POSIX.1)
EBADR	Invalid request descriptor
EBADRQC	Invalid request code

EBADSLT
Invalid slot

EBUSY
Device or resource busy (POSIX.1)

ECANCELED
Operation canceled (POSIX.1)

ECHILD
No child processes (POSIX.1)

ECHRNG
Channel number out of range

ECOMM
Communication error on send

ECONNABORTED
Connection aborted (POSIX.1)

ECONNREFUSED
Connection refused (POSIX.1)

ECONNRESET
Connection reset (POSIX.1)

EDEADLK
Resource deadlock avoided (POSIX.1)

EDEADLOCK
Synonym for **EDEADLK**

EDESTADDRREQ
Destination address required (POSIX.1)

EDOM
Mathematics argument out of domain of function (POSIX.1, C99)

EDQUOT
Disk quota exceeded (POSIX.1)

EEXIST
File exists (POSIX.1)

EFAULT
Bad address (POSIX.1)

EFBIG
File too large (POSIX.1)

EHOSTDOWN
Host is down

EHOSTUNREACH
Host is unreachable (POSIX.1)

EIDRM
Identifier removed (POSIX.1)

EILSEQ
Illegal byte sequence (POSIX.1, C99)

EINPROGRESS
Operation in progress (POSIX.1)

EINTR
Interrupted function call (POSIX.1)

EINVAL
Invalid argument (POSIX.1)

EIO Input/output error (POSIX.1)

EISCONN
Socket is connected (POSIX.1)

EISDIR
Is a directory (POSIX.1)

EISNAM
Is a named type file

EKEYEXPIRED
Key has expired

EKEYREJECTED
Key was rejected by service

EKEYREVOKED
Key has been revoked

EL2HLT
Level 2 halted

EL2NSYNC
Level 2 not synchronized

EL3HLT
Level 3 halted

EL3RST
Level 3 halted

ELIBACC
Cannot access a needed shared library

ELIBBAD
Accessing a corrupted shared library

ELIBMAX
Attempting to link in too many shared libraries

ELIBSCN
lib section in a.out corrupted

ELIBEXEC
Cannot exec a shared library directly

ELOOP
Too many levels of symbolic links (POSIX.1)

EMEDIUMTYPE
Wrong medium type

EMFILE
Too many open files (POSIX.1)

EMLINK
Too many links (POSIX.1)

EMSGSIZE
Message too long (POSIX.1)

EMULTIHOP
Multihop attempted (POSIX.1)

ENAMETOOLONG
Filename too long (POSIX.1)

ENETDOWN
Network is down (POSIX.1)

ENETRESET
Connection aborted by network (POSIX.1)

ENETUNREACH
Network unreachable (POSIX.1)

ENFILE
Too many open files in system (POSIX.1)

ENOBUFS
No buffer space available (POSIX.1 (XSI STREAMS option))

ENODATA
No message is available on the STREAM head read queue (POSIX.1)

ENODEV
No such device (POSIX.1)

ENOENT
No such file or directory (POSIX.1)

ENOEXEC
Exec format error (POSIX.1)

ENOKEY
Required key not available

ENOLCK
No locks available (POSIX.1)

ENOLINK
Link has been severed (POSIX.1)

ENOMEDIUM
No medium found

ENOMEM
Not enough space (POSIX.1)

ENOMSG
No message of the desired type (POSIX.1)

ENONET
Machine is not on the network

ENOPKG
Package not installed

ENOPROTOOPT
Protocol not available (POSIX.1)

ENOSPC
No space left on device (POSIX.1)

ENOSR
No STREAM resources (POSIX.1 (XSI STREAMS option))

ENOSTR
Not a STREAM (POSIX.1 (XSI STREAMS option))

ENOSYS
Function not implemented (POSIX.1)

ENOTBLK
Block device required

ENOTCONN
The socket is not connected (POSIX.1)

ENOTDIR
Not a directory (POSIX.1)

ENOTEMPTY
Directory not empty (POSIX.1)

ENOTSOCK
Not a socket (POSIX.1)

ENOTSUP
Operation not supported (POSIX.1)

ENOTTY
Inappropriate I/O control operation (POSIX.1)

ENOTUNIQ
Name not unique on network

ENXIO
No such device or address (POSIX.1)

EOPNOTSUPP
Operation not supported on socket (POSIX.1)

(ENOTSUP and EOPNOTSUPP have the same value on Linux, but according to POSIX.1 these error values should be distinct.)

EOVERFLOW
Value too large to be stored in data type (POSIX.1)

EPERM
Operation not permitted (POSIX.1)

EPFNOSUPPORT
Protocol family not supported

EPIPE Broken pipe (POSIX.1)

EPROTO
Protocol error (POSIX.1)

EPROTONOSUPPORT
Protocol not supported (POSIX.1)

EPROTOTYPE
Protocol wrong type for socket (POSIX.1)

ERANGE
Result too large (POSIX.1, C99)

EREMCHG
Remote address changed

EREMOTE
Object is remote

EREMOTEIO
Remote I/O error

ERESTART
Interrupted system call should be restarted

EROFS
Read-only file system (POSIX.1)

ESHUTDOWN
Cannot send after transport endpoint shutdown

ESPIPE
Invalid seek (POSIX.1)

ESOCKTNOSUPPORT
Socket type not supported

ESRCH
No such process (POSIX.1)

ESTALE
Stale file handle (POSIX.1))

This error can occur for NFS and for other file systems

ESTRPIPE
Streams pipe error

ETIME
Timer expired (POSIX.1 (XSI STREAMS option))

(POSIX.1 says "STREAM **ioctl**() timeout")

ETIMEDOUT
Connection timed out (POSIX.1)

ETXTBSY
Text file busy (POSIX.1)

EUCLEAN
Structure needs cleaning

EUNATCH
Protocol driver not attached

EUSERS
Too many users

EWOULDBLOCK
Operation would block (may be same value as **EAGAIN**) (POSIX.1)

EXDEV
Improper link (POSIX.1)

EXFULL
Exchange full

NOTES

A common mistake is to do

```
if (somecall() == −1) {
    printf("somecall() failed\n");
    if (errno == ...) { ... }
}
```

where *errno* no longer needs to have the value it had upon return from *somecall*() (i.e., it may have been changed by the **printf**()). If the value of *errno* should be preserved across a library call, it must be saved:

```
if (somecall() == −1) {
    int errsv = errno;
    printf("somecall() failed\n");
    if (errsv == ...) { ... }
}
```

It was common in traditional C to declare *errno* manually (i.e., *extern int errno*) instead of including <*errno.h*>. **Do not do this**. It will not work with modern versions of the C library. However, on (very) old Unix systems, there may be no <*errno.h*> and the declaration is needed.

SEE ALSO

err(3), **error**(3), **perror**(3), **strerror**(3)

NAME

wait, waitpid − wait for process to change state

SYNOPSIS

#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int ***status*****);**
pid_t waitpid(pid_t *pid***, int** ***status*****, int** *options***);**
int waitid(idtype_t *idtype***, id_t** *id***, siginfo_t** ***infop*****, int** *options***);**

DESCRIPTION

All of these system calls are used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed. A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal. In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then terminated the child remains in a "zombie" state (see NOTES below).

If a child has already changed state, then these calls return immediately. Otherwise they block until either a child changes state or a signal handler interrupts the call (assuming that system calls are not automatically restarted using the **SA_RESTART** flag of **sigaction**(2)). In the remainder of this page, a child whose state has changed and which has not yet been waited upon by one of these system calls is termed *waitable*.

wait() and waitpid()

The **wait**() system call suspends execution of the current process until one of its children terminates. The call *wait(&status)* is equivalent to:

```
waitpid(−1, &status, 0);
```

The **waitpid**() system call suspends execution of the current process until a child specified by *pid* argument has changed state. By default, **waitpid**() waits only for terminated children, but this behaviour is modifiable via the *options* argument, as described below.

The value of *pid* can be:

< −1	meaning wait for any child process whose process group ID is equal to the absolute value of *pid*.
−1	meaning wait for any child process.
0	meaning wait for any child process whose process group ID is equal to that of the calling process.
> 0	meaning wait for the child whose process ID is equal to the value of *pid*.

The value of *options* is an OR of zero or more of the following constants:

WNOHANG
return immediately if no child has exited.

WUNTRACED
also return if a child has stopped (but not traced via **ptrace**(2)). Status for *traced* children which have stopped is provided even if this option is not specified.

WCONTINUED
(Since Linux 2.6.10) also return if a stopped child has been resumed by delivery of **SIGCONT**.

(For Linux-only options, see below.)

The **WUNTRACED** and **WCONTINUED** options are only effective if the **SA_NOCLDSTOP** flag has not been set for the **SIGCHLD** signal (see **sigaction**(2)).

If *status* is not NULL, **wait**() and **waitpid**() store status information in the *int* to which it points. This integer can be inspected with the following macros (which take the integer itself as an argument, not a pointer to it, as is done in **wait**() and **waitpid**()!):

WIFEXITED(*status*)
returns true if the child terminated normally, that is, by calling **exit**(3) or **_exit**(2), or by returning from main().

WEXITSTATUS(*status*)
returns the exit status of the child. This consists of the least significant 16-8 bits of the *status* argument that the child specified in a call to **exit**() or **_exit**() or as the argument for a return statement in main(). This macro should only be employed if **WIFEXITED** returned true.

WIFSIGNALED(*status*)
returns true if the child process was terminated by a signal.

WTERMSIG(*status*)
returns the number of the signal that caused the child process to terminate. This macro should only be employed if **WIFSIGNALED** returned true.

WCOREDUMP(*status*)
returns true if the child produced a core dump. This macro should only be employed if **WIFSIGNALED** returned true. This macro is not specified in POSIX.1-2001 and is not available on some Unix implementations (e.g., AIX, SunOS). Only use this enclosed in #ifdef WCOREDUMP ... #endif.

WIFSTOPPED(*status*)
returns true if the child process was stopped by delivery of a signal; this is only possible if the call was done using **WUNTRACED** or when the child is being traced (see **ptrace**(2)).

WSTOPSIG(*status*)
returns the number of the signal which caused the child to stop. This macro should only be employed if **WIFSTOPPED** returned true.

WIFCONTINUED(*status*)
(Since Linux 2.6.10) returns true if the child process was resumed by delivery of **SIGCONT**.

waitid()

The **waitid**() system call (available since Linux 2.6.9) provides more precise control over which child state changes to wait for.

The *idtype* and *id* arguments select the child(ren) to wait for, as follows:

idtype == **P_PID**
Wait for the child whose process ID matches *id*.

idtype == **P_PGID**
Wait for any child whose process group ID matches *id*.

idtype == **P_ALL**
Wait for any child; *id* is ignored.

The child state changes to wait for are specified by ORing one or more of the following flags in *options*:

WEXITED
Wait for children that have terminated.

WSTOPPED
Wait for children that have been stopped by delivery of a signal.

WCONTINUED
Wait for (previously stopped) children that have been resumed by delivery of **SIGCONT**.

The following flags may additionally be ORed in *options*:

WNOHANG
As for **waitpid**().

WNOWAIT
Leave the child in a waitable state; a later wait call can be used to again retrieve the child status information.

Upon successful return, **waitid**() fills in the following fields of the *siginfo_t* structure pointed to by *infop*:

si_pid The process ID of the child.

si_uid The real user ID of the child. (This field is not set on most other implementations.)

si_signo
Always set to **SIGCHLD**.

si_status
Either the exit status of the child, as given to **_exit**(2) (or **exit**(3)), or the signal that caused the child to terminate, stop, or continue. The *si_code* field can be used to determine how to interpret this field.

si_code Set to one of: **CLD_EXITED** (child called **_exit**(2)); **CLD_KILLED** (child killed by signal); **CLD_STOPPED** (child stopped by signal); or **CLD_CONTINUED** (child continued by **SIG-CONT**).

If **WNOHANG** was specified in *options* and there were no children in a waitable state, then **waitid**() returns 0 immediately and the state of the *siginfo_t* structure pointed to by *infop* is unspecified. To distinguish this case from that where a child was in a waitable state, zero out the *si_pid* field before the call and check for a non-zero value in this field after the call returns.

RETURN VALUE

wait(): on success, returns the process ID of the terminated child; on error, −1 is returned.

waitpid(): on success, returns the process ID of the child whose state has changed; on error, −1 is returned; if **WNOHANG** was specified and no child(ren) specified by *pid* has yet changed state, then 0 is returned.

waitid(): returns 0 on success or if **WNOHANG** was specified and no child(ren) specified by *id* has yet changed state; on error, −1 is returned.

Each of these calls sets *errno* to an appropriate value in the case of an error.

ERRORS

ECHILD
(for **wait**()) The calling process does not have any unwaited-for children.

ECHILD
(for **waitpid**() or **waitid**()) The process specified by *pid* (**waitpid**()) or *idtype* and *id* (**waitid**()) 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. See also the LINUX NOTES section about threads.)

EINTR
WNOHANG was not set and an unblocked signal or a **SIGCHLD** was caught.

EINVAL
The *options* argument was invalid.

NOTES

A child that terminates, but has not been waited for becomes a "zombie". The kernel maintains a minimal set of information about the zombie process (PID, termination status, resource usage information) in order to allow the parent to later perform a wait to obtain information about the child. As long as a zombie is not removed from the system via a wait, it will consume a slot in the kernel process table, and if this table fills, it will not be possible to create further processes. If a parent process terminates, then its "zombie" children (if any) are adopted by **init**(8), which automatically performs a wait to remove the zombies.

POSIX.1-2001 specifies that if the disposition of **SIGCHLD** is set to **SIG_IGN** or the **SA_NOCLDWAIT** flag is set for **SIGCHLD** (see **sigaction**(2)), then children that terminate do not become zombies and a call to **wait**() or **waitpid**() will block until all children have terminated, and then fail with *errno* set to **ECHILD**. (The original POSIX standard left the behaviour of setting **SIGCHLD** to **SIG_IGN** unspecified.) Linux 2.6 conforms to this specification. However, Linux 2.4 (and earlier) does not: if a **wait**() or **waitpid**() call is made while **SIGCHLD** is being ignored, the call behaves just as though **SIGCHLD** were not being ignored, that is, the call blocks until the next child terminates and then returns the process ID and status of that child.

LINUX NOTES

In the Linux kernel, a kernel-scheduled thread is not a distinct construct from a process. Instead, a thread is simply a process that is created using the Linux-unique **clone**(2) system call; other routines such as the portable **pthread_create**(3) call are implemented using **clone**(2). Before Linux 2.4, a thread was just a special case of a process, and as a consequence one thread could not wait on the children of another thread, even when the latter belongs to the same thread group. However, POSIX prescribes such functionality, and since Linux 2.4 a thread can, and by default will, wait on children of other threads in the same thread group.

The following Linux-specific *options* are for use with children created using **clone**(2); they cannot be used with **waitid**():

__WCLONE
: Wait for "clone" children only. If omitted then wait for "non-clone" children only. (A "clone" child is one which delivers no signal, or a signal other than **SIGCHLD** to its parent upon termination.) This option is ignored if **__WALL** is also specified.

__WALL
: (Since Linux 2.4) Wait for all children, regardless of type ("clone" or "non-clone").

__WNOTHREAD
: (Since Linux 2.4) Do not wait for children of other threads in the same thread group. This was the default before Linux 2.4.

EXAMPLE

The following program demonstrates the use of **fork(2)** and **waitpid**(2). The program creates a child process. If no command-line argument is supplied to the program, then the child suspends its execution using **pause**(2), to allow the user to send signals to the child. Otherwise, if a command-line argument is supplied, then the child exits immediately, using the integer supplied on the command line as the exit status. The parent process executes a loop that monitors the child using **waitpid**(2), and uses the W*() macros described above to analyse the wait status value.

The following shell session demonstrates the use of the program:

```
$ ./a.out &
Child PID is 32360
[1] 32359
$ kill -STOP 32360
stopped by signal 19
$ kill -CONT 32360
continued
$ kill -TERM 32360
killed by signal 15
[1]+  Done                    ./a.out
$
```

```
#include <sys/wait.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
```

```
int
main(int argc, char *argv[])
{
    pid_t cpid, w;
    int status;

    cpid = fork();
    if (cpid == -1) { perror("fork"); exit(EXIT_FAILURE); }

    if (cpid == 0) {            /* Code executed by child */
        printf("Child PID is %ld\n", (long) getpid());
        if (argc == 1)
            pause();                    /* Wait for signals */
        _exit(atoi(argv[1]));

    } else {                    /* Code executed by parent */
        do {
            w = waitpid(cpid, &status, WUNTRACED | WCONTINUED);
            if (w == -1) { perror("waitpid"); exit(EXIT_FAILURE); }

            if (WIFEXITED(status)) {
                printf("exited, status=%d\n", WEXITSTATUS(status));
            } else if (WIFSIGNALED(status)) {
                printf("killed by signal %d\n", WTERMSIG(status));
            } else if (WIFSTOPPED(status)) {
                printf("stopped by signal %d\n", WSTOPSIG(status));
            } else if (WIFCONTINUED(status)) {
                printf("continued\n");
            }
        } while (!WIFEXITED(status) && !WIFSIGNALED(status));
        exit(EXIT_SUCCESS);
    }
}
```

CONFORMING TO

SVr4, 4.3BSD, POSIX.1-2001.

SEE ALSO

_exit(2), **clone**(2), **fork**(2), **kill**(2), **ptrace**(2), **sigaction**(2), **signal**(2), **wait4**(2), **pthread_create**(3), **signal**(7)

NAME

chdir, fchdir − change working directory

SYNOPSIS

```
#include <unistd.h>

int chdir(const char *path);
int fchdir(int fd);
```

DESCRIPTION

chdir() changes the current working directory to that specified in *path*.

fchdir() is identical to **chdir**(); the only difference is that the directory is given as an open file descriptor.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set appropriately.

ERRORS

Depending on the file system, other errors can be returned. The more general errors for **chdir**() are listed below:

EACCES
Search permission is denied for one of the directories in the path prefix of *path*. (See also **path_resolution**(2).)

EFAULT
path points outside your accessible address space.

EIO An I/O error occurred.

ELOOP
Too many symbolic links were encountered in resolving *path*.

ENAMETOOLONG
path is too long.

ENOENT
The file does not exist.

ENOMEM
Insufficient kernel memory was available.

ENOTDIR
A component of *path* is not a directory.

The general errors for **fchdir**() are listed below:

EACCES
Search permission was denied on the directory open on *fd*.

EBADF
fd is not a valid file descriptor.

NOTES

A child process created via **fork**(2) inherits its parent's current working directory. The current working directory is left unchanged by **execve**(2).

The prototype for **fchdir**() is only available if **_BSD_SOURCE** is defined, or **_XOPEN_SOURCE** is defined with the value 500.

CONFORMING TO

SVr4, 4.4BSD, POSIX.1-2001.

SEE ALSO

chroot(2), **path_resolution**(2), **getcwd**(3)