POSIX, Part 1: Error handling

Edit New Page

SudarshanGp edited this page on Apr 8 · 3 revisions

What is `errno` and when is it set?

POSIX defines a special integer `errno` that is set when a system call fails. The initial value of `errno` is zero (i.e. no error). When a system call fails it will typically return -1 to indicate an error and set `errno`

What about multiple threads?

Each thread has it's own copy of `errno`. This is very useful; otherwise an error in one thread would interfere with the error status of another thread.

When is `errno` reset to zero?

It's not unless you specifically reset it to zero! When system calls are successful they do *not* reset the value of `errno`.

This means you should only rely on the value of errno if you know a system call has failed (e.g. it returned -1).

What are the gotchas and best practices of using `errno`?

Be careful when complex error handling use of library calls or system calls that may change the value of `errno`. In practice it's safer to copy the value of errno into a int variable:

```
// Unsafe - the first fprintf may change the value of errno before we use it!
if (-1 == sem_wait(&s)) {
   fprintf(stderr, "An error occurred!");
   fprintf(stderr, "The error value is %d\n", errno);
}
// Better, copy the value before making more system and library calls
if (-1 == sem_wait(&s)) {
   int errno_saved = errno;
   fprintf(stderr, "An error occurred!");
   fprintf(stderr, "The error value is %d\n", errno_saved);
}
```

In a similar vein, if your signal handler makes any system or library calls, then it is good practice to save the original value of errno and restore the value before returning:

Pages 51

Find a Page…

Home

#Example Markdown

#Informal Glossary

#Piazza: When And How to Ask For Help

C Programming, Part 1: Introduction

C Programming, Part 2: Text Input And Output

C Programming, Part 3: Common Gotchas

C Programming, Part 4: Debugging

Deadlock, Part 1: Resource Allocation Graph

Deadlock, Part 2: Deadlock Conditions

File System, Part 1: Introduction

File System, Part 2: Files are inodes (everything else is just data...)

File System, Part 3: Permissions

File System, Part 4: Working with directories

File System, Part 5: Virtual file systems

Show 36 more pages…

Clone this wiki locally

https://github.com/angrave/SystemPr

Clone in Desktop

```
void handler(int signal) {
   int errno_saved = errno;

   // make system calls that might change errno

   errno = errno_saved;
}
```

How can you print out the string message associated with a particular error number?

Use `strerror` to get a short (English) description of the error value

```
char *mesg = strerror(errno);
fprintf(stderr, "An error occurred (errno=%d): %s", errno, mesg);
```

How are perror and strerror related?

In previous pages we've used perror to print out the error to standard error. Using `strerror`, we can now write a simple implementation of `perror`:

```
void perror(char *what) {
   fprintf(stderr, "%s: %s\n", what, strerror(errno));
}
```

What are the gotchas of using strerror?

Unfortunately `strerror` is not threadsafe. In other words, two threads cannot call it at the same time!

There are two workarounds: Firstly we can use a mutex lock to define a critical section and a local buffer. The same mutex should be used by all threads in all places that call `strerror`

```
pthread_mutex_lock(&m);
char *result = strerror(errno);
char *message = malloc(strlen(result) + 1);
strcpy(message, result);
pthread_mutex_unlock(&m);
fprintf(stderr, "An error occurred (errno=%d): %s", errno, message);
free(message);
```

Alternatively use the less portable but thread-safe `strerror_r`

What is EINTR? What does it mean for sem_wait? read? write?

Some system calls can be interrupted when a signal (e.g SIGCHLD, SIGPIPE,...) is delivered to the process. At this point the system call may return without performing any action! For example, bytes may not have been read/written, semaphore wait may not have waited.

This interruption can be detected by checking the return value and if `errno` is EINTR. In which case the system call should be retried. It's common to see the following kind of loop that wraps a system call (such as sem_wait).

```
while ((-1 == systemcall(...)) && (errno == EINTR)) { /* repeat! */}
```

Be careful to write `== EINTR`, not `= EINTR`.

Or, if the result value needs to be used later...

```
while ((-1 == (result = systemcall(...))) && (errno == EINTR)) { /* repeat! */}
```

On Linux,calling `read` and `write` to a local disk will normally not return with EINTR (instead the function is automatically restarted for you). However, calling `read` and `write` on a file descriptor that corresponds to a network stream *can* return with EINTR.

Which system calls may be interrupted and need to be wrapped?

Use man the page! The man page includes a list of errors (i.e. errno values) that may be set by the system call. A rule of thumb is 'slow' (blocking) calls (e.g. writing to a socket) may be interrupted but fast non-blocking calls (e.g. pthread_mutex_lock) will not.

From the linux signal 7 man page.

"If a signal handler is invoked while a system call or library function call is blocked, then either:

- the call is automatically restarted after the signal handler returns; or
- the call fails with the error EINTR. Which of these two behaviors occurs depends on the interface and whether or not the signal handler was established using the SA_RESTART flag (see sigaction(2)). The details vary across UNIX systems; below, the details for Linux.

If a blocked call to one of the following interfaces is interrupted by a signal handler, then the call will be automatically restarted after the signal handler returns if the SA_RESTART flag was used; otherwise the call will fail with the error EINTR:

- read(2), readv(2), write(2), writev(2), and ioctl(2) calls on "slow" devices. A "slow" device is one where the I/O call may block for an indefinite time, for example, a terminal, pipe, or socket. (A disk is not a slow device according to this definition.) If an I/O call on a slow device has already transferred some data by the time it is interrupted by a signal handler, then the call will return a success status (normally, the number of bytes transferred). "

Note, it is easy to believe that setting 'SA_RESTART' flag is sufficient to make this whole problem disappear. Unfortunately that's not true: there are still system calls that may return early and set `EINTR`! See signal(7) for details.