Kernel dependencies

The biggest single problem in porting software is the operating system. The operating system services play a large part in determining how a program must be written. UNIX versions differ enough in some areas to require significant modifications to programs to adapt them to a different version. In this and the following chapters, we'll look at what has happened to UNIX since it was essentially a single system, round the time of the Seventh Edition.

Many books have been written on the internals of the various UNIX flavours, for example *The Design of the UNIX System* by Maurice Bach for System V.2, *The Design and the Implementation of the 4.3BSD UNIX Operating System* by Sam Leffler, Kirk McKusick, Mike Karels, and John Quarterman for 4.3BSD, and *The Magic Garden explained: The Internals of UNIX System V Release 4* by Berny Goodheart and James Cox for System V.4. In addition, a number of books have been written about programming in these environments—*Advanced Programming in the UNIX environment* by Richard Stevens gives an excellent introduction to System V.4 and "4.3+BSD"[*] for programmers. In this chapter and the ones following it, we'll restrict our view to brief descriptions of aspects that can cause problems when porting software from one UNIX platform to another. We'll look at specific areas in Chapter 12, *Kernel dependencies*, Chapter 13, *Signals*, Chapter 14, *File systems* and Chapter 15, *Terminal drivers*. In the rest of this chapter, we'll look at:

- Interprocess communication
- Non-blocking I/O
- Miscellaneous aspects of kernel functionality

The descriptions are not enough to help you use the functionality in writing programs: they are intended to help you understand existing programs and rewrite them in terms of functions available to you. If you need more information, you may find it in the 4.4BSD man pages (see Appendix E, *Where to get sources*), or in *Advanced Programming in the UNIX environment*, by Richard Stevens.

* 4.3BSD was released in 1987, 4.4BSD in 1994. In the time in between, releases had names like *4.3BSD Tahoe*, *4.3BSD Reno*, and *NET/2*. For want of a better term, Stevens refers to systems roughly corresponding to NET/2 as *4.3+BSD*.

Interprocess communication

interprocess communication (frequently written as the abbreviation *IPC*), the ability to transfer data between processes, was one of the important original concepts of UNIX. The original methods were what you might expect of a concept that, at the time, was revolutionary and still under development: there were more than a few limitations. Even today there is no agreement on how interprocess communication should take place.

In this section we'll look very briefly at the various kinds of interprocess communication, and what to do if the package you are porting uses a method your kernel doesn't support. To start with the bad news: if you find your kernel doesn't support the IPC model that the package expects, you will probably need to make significant modifications to adapt it to a different model.

Interprocess communication was originally limited to a single processor, but of course network communication is also a form of interprocess communication. We'll touch briefly on network communication in the following discussion.

UNIX systems offer a bewildering number of different forms of interprocess communication:

- *Pipes* are the original form of communication and are found in all versions of UNIX. They have the disadvantages that they transfer data in one direction only, and that they can only connect two processes that have a common ancestor.
- *Sockets* are the BSD interprocess communication mechanism: they are by far the most powerful mechanism, offering unidirectional, bidirectional and network communication. In BSD systems, they are even used to implement the `pipe` system call.
- *STREAMS*[*] is a generalized I/O concept available in newer System V systems and their derivatives. It was originally intended to replace character device drivers, but its flexibility makes it useful for interprocess communication as well. Like sockets, it can be used both for local and remote communication. *UNIX Network Programming*, by Richard Stevens, describes STREAMS in some detail, and *The Magic Garden Explained* describes the implementation. We won't consider them further here.
- *Stream pipes* differ from normal pipes by being able to transfer data in both directions. They have no particular connection with STREAMS.
- *FIFOs*, also called *named pipes*, are like pipes, but they have a name in the file system hierarchy.
- *Named stream pipes* are stream pipes with names. They bear the same relationship to stream pipes that FIFOs do to normal pipes.
- *System V IPC* is a bundle that offers *message queues*, yet another form of message passing, *shared memory*, which enables processes to pass data directly, and *semaphores*, which synchronize processes.

* Why the shouting? STREAMS was derived from the Eighth Edition *Streams* concept (see *S Stream Input-Output System*, by Dennis Ritchie). System V always spells it in upper case, so this is a convenient way of distinguishing between the implementations.

In the following sections, we'll look at these features in a little more detail.

Pipes

The original UNIX interprocess communication facility was *pipes*. Pipes are created by the *pipe* function call:

```
#include <unistd.h>

int pipe (int *fildes);
```

This call creates a pipe with two file descriptors, a read descriptor and a write descriptor. It returns the value of the read descriptor to `fildes [0]` and the value of the write descriptor to `fildes [1]`. At this point, only the creating process can use the pipe, which is not very useful. After calling `fork`, however, both of the resultant processes can use the pipe. Depending on their purpose, the processes may decide to close one direction of the pipe: for example, if you write output to the *more* program, you don't expect any reply from *more*, so you can close the read file descriptor.

A fair amount of code is involved in opening a pipe, starting a new process with `fork` and `exec` and possibly waiting for it terminate with `wait`. The standard library functions `popen` and `pclose` make this job easier:

```
#include <stdio.h>

FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

`popen` creates a pipe, then forks and execs a shell with `command` as its parameter. `type` specifies whether the pipe should be open for reading ("`r`") or writing ("`w`"). Since pipes are unidirectional, they cannot be opened both for reading and for writing.

After opening the command, you can write to the process with normal `write` commands. On completion, `pclose` waits for the child process to terminate and closes the file descriptors.

Sockets

Sockets were originally developed at Berkeley as part of the TCP/IP networking implementation introduced with 4.2BSD, but they are in fact a general interprocess communication facility. In BSD systems, the other interprocess communication facilities are based on sockets.

Most of the features of sockets are related to networking, which we don't discuss here. The call is:

```
#include <sys/types.h>
#include <sys/socket.h>

int socket (int domain, int type, int protocol);
```

- *domain* specifies the communications domain. Common domains are `AF_UNIX` (UNIX

domain),* used for local communication, `AF_INET` (Internet domain), and `AF_ISO` (ISO protocol domain).

- `type` specifies the type of socket. For local interprocess communication, you would use *SOCK_STREAM*, which supplies a reliable two-way communication channel.
- `protocol` specifies the communications protocol to use. In the UNIX domain, this parameter is not used and should be set to 0.

As we shall see in the next section, the way that pipes are implemented means that you need two sockets to simulate a pipe. You can do this with the *socketpair* system call, which creates a pair of file descriptors with identical properties.

```
#include <sys/types.h>
#include <sys/socket.h>

int socketpair (int domain, int type, int protocol, int *sv);
```

Currently, `socketpair` works only in the UNIX domain, so you don't have much choice in the parameters: `domain` must be `AF_UNIX`, `type` must be `SOCK_STREAM`, and `protocol` is meaningless in the UNIX domain. The only important parameter is `sv`, which is where the socket descriptors are returned—exactly the same thing as the `fildes` parameter to `pipe`.

Most systems have some kind of socket support, but sometimes it is just an emulation library that omits significant functionality, such as the UNIX domain and the `socketpair` call. Many older System V sockets emulation libraries also have a bad reputation regarding performance and reliability. On the other hand, many System V.3 ports included the original Berkeley socket implementation in the kernel.

Other kinds of pipe

Pipes have two main restrictions:

- They are unidirectional: you write to one descriptor, you read from the other. It would be a nice idea to be able to read from and write to the same descriptor.
- They are anonymous: you don't open an existing pipe, you create a new one, and only you and your descendents can use it. It would be nice to be able to use pipes like regular files.

In fact, you can get all combinations of these properties. We've seen regular pipes—the others are *stream pipes*, *FIFOs* and *named stream pipes*. We'll look at them in the following sections:

* Not all UNIX implementations support UNIX domain sockets. In particular, some System V systems support only the Internet domain. People with a System V background often place the emphasis on the word "domain", and some even refer to UNIX domain sockets as "domain sockets". As you can see from the above, this is incorrect.

Stream pipes

Most systems allow you to create bidirectional pipes. For some reason, they're generally called *stream pipes*, which is not a good name at all.

- In System V.4, regular pipes are bi-directional, so you don't need to do anything special.
- In 4.4BSD, the *socketpair* system call, which we have already seen, creates stream pipes, so you'd expect regular pipes to be bidirectional in 4.4BSD as well. In fact, before returning, the library function *pipe* closes one descriptor in each direction, so 4.4BSD pipes really are unidirectional. If you want a stream pipe, just use the `socketpair` system call.
- In System V.3 systems with STREAMS, bidirectional pipes are possible too, but things are more difficult: you have to connect two streams back to back. See *UNIX Network Programming* for a discussion of how to do this.

FIFOs

*FIFO*s are pipes with file names, which allow unrelated processes to communicate with each other. To create a FIFO, you use the function `mkfifo`:

```
#include <sys/stat.h>

int mkfifo (const char *path, mode_t mode);
```

This call corresponds exactly to `mkdir`, except that it creates a FIFO instead of a directory. BSD implements `mkfifo` as a system call, while System V.4 implements it as a library function that calls `mknod` to do the work. System V.3 systems frequently implemented it as a system call. Once you have created a FIFO, you can use it just like a file: typically, one process, the *listener process*, opens the FIFO for reading, and one or more open it for writing to the listener process.

Named stream pipes

Stream pipes are bidirectional, but they don't normally have names. FIFOs have names, but they're usually not bidirectional. To get both of these properties, we need a new kind of connection, a *named stream pipe*. In 4.4BSD, this can be done by binding a name to a stream socket—see the man pages for `bind` for further details. In System V.4, you can create a named stream pipe with the `connld` STREAMS module. See *Advanced Programming in the UNIX environment* for more details.

System V IPC

System V supplies an alternative form of interprocess communication consisting of three features: *shared memory*, *message queues* and *semaphores*. SunOS 4 also supports System V IPC, but pure BSD systems do not. In the industry there is a significant amount of aversion to this implementation, which is sometimes called *The Three Ugly Sisters*.

System V IPC is overly complicated and sensitive to programming bugs, which are two of the main reasons why it has not been implemented on other systems. Converting programs written for System V IPC to other methods of interprocess communication is non-trivial. If you have a BSD system with kernel sources, it might be easier to implement Daniel Boulet's free software implementation (see Appendix E, *Where to get sources*).

Shared memory

An alternative form of interprocess communication involves sharing data between processes. Instead of sending a message, you just write it into a buffer that is also mapped into the address space of the other process. There are two forms of shared memory that you may encounter on UNIX systems—System V shared memory and `mmap`, which is more commonly used for mapping files to memory. We'll look at `mmap` in Chapter 14, *File systems*, page 232.

System V shared memory is implemented with four system calls:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget (key_t key, int size, int shmflg);
int shmctl (int shmid, int cmd, ...     /* struct shmid_ds *buf */);
void *shmat (int shmid, void *shmaddr, int shmflg);
int shmdt (void *shmaddr);
```

- `shmget` allocates a shared memory segment or adds the process to the list of processes sharing the segment. The shared memory segment identifier is conceptually like a file name or an identifier, but for some reason they are called `keys` when talking about System V shared memory. It returns a segment identifier, conceptually like a file number.
- `shmctl` performs control operations on shared memory segments. It can set ownerships and permissions, retrieve status information, or remove shared memory segments. Like files, shared memory segments remain on the system until explicitly removed, even if they are currently not assigned to any process.
- `shmat` attaches the shared memory segment `shmid` to the calling process.
- `shmdt` detaches a shared memory segment from the calling process.

With some limitations, you can use `mmap` to replace System V shared memory. The limitations are that `mmap` on non-System V platforms normally maintains separate data pages for each process, so if you write to a page in one process, other processes will not see the new data. You need to call `msync` in order to update the segments used by other processes. Between the time when you modify the segment and when you call `msync`, the data is inconsistent. `msync` is not a fast call, so this could also cripple performance.

Message queues

As if there weren't enough ways of passing data between processes already, System V IPC includes *message queues*. Message queues are rather like FIFOs, but there are two differences:

- A FIFO transmits a byte stream, but a message queue is record oriented.
- Messages can have different priorities, which determine the sequence in which they are received, if the receiving process allows them to queue up.

The system calls to handle message queues are analogous to the shared memory calls:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget (key_t key, int msgflg);
int msgsnd (int msqid, const void *msgp, size_t msgsz, int msgflg);
int msgrcv (int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
int msgctl (int msqid, int cmd, .../* struct msqid_ds *buf */);
```

- `msgget` opens an existing queue or creates a new queue.
- `msgsnd` sends a message.
- `msgrcv` receives a message.
- `msgctl` performs control functions on message queues.

Message queues were originally intended to offer fast interprocess communication. Nowadays they have little to offer that a FIFO couldn't handle. If you run into problems with message queues, you might prefer to replace them with FIFOs.

Semaphores

One disadvantage with shared memory implementations is that one process doesn't know when another process has done something. This can have a number of consequences:

- Two processes may modify the same area at the same time.
- One process may be waiting for the other process to do something, and needs to know when it has finished.

There are two possible solutions: send a signal, or use *semaphores*.

A semaphore is a means of voluntary process synchronization, similar to advisory locking. To use the facility, a process requests access to the semaphore. If access is currently not possible, the process blocks until access is permitted. Unlike locking, semaphores allow more than one process access to the semaphore at any one point. They do this by maintaining a counter, a small positive integer, in the semaphore. When a process accesses the semaphore, it decrements the counter. If the value of the counter is still non-negative, the process has access, otherwise it is blocked. This could be used to gain access to a limited number of resources.

System V semaphores look superfi cially similar to System V shared memory. There are three functions:

```
int semctl (int semid, int semnum, int cmd, ... /* union semun arg */);
int semget (key_t key, int nsems, int semflg);
int semop (int semid, struct sembuf *sops, size_t nsops);
```

The implementation is less than perfect. In particular, it is overly complex, and it almost encourages deadlocks, situations where no process can continue:

- Instead of a single counter, a System V semaphore declares an array of counters. The size of the array is determined by the nsems parameter of the semget system call.
- It takes two calls (semget and semctl) to create and initialize a semaphore. Theoretically, this creates an opportunity for another process to come and initialize the semaphore differently.
- It's possible for semaphores to remain locked after a process ends, which means that a reboot is necessary to unlock the semaphore again. A flag is provided to specify that a semaphore should be removed on exit, but you can't rely upon it completely.
- The implementation is not very fast.

Miscellaneous system functionality

The rest of this chapter describes miscellaneous system calls that can occasionally cause problems when porting.

exec

exec is one of the original system calls at the heart of the UNIX system, so it may come as a surprise to discover that exec is no longer a system call on modern systems—instead, it is implemented as a library function in terms of new system calls such as execve. Even the Seventh Edition man pages stated

> *Plain* exec *is obsoleted by* execve*, but remains for historical reasons.*

Nowadays, there are a large number of alternatives. Your system probably has most of the following calls:

```
#include <unistd.h>
extern char **environ;

int exec    (char *path, char *argv []);
int execve  (char *path, char *argv [], char *envp []);
int execl   (char *path, char *arg, ..., NULL);
int execle  (char *path, char *arg, ..., NULL, char *envp []);
int execlp  (char *file, char *arg, ..., NULL);
int execlpe (char *file, char *arg, ..., NULL, char *envp []);
int exect   (char *path, char *argv [], char *envp []);
```

```
int execv   (char *path, char *argv []);
int execve  (char *path, char *argv [], char *envp []);
int execvp  (char *file, char *argv []);
int execvpe (char *file, char *argv [], char *envp []);
```

All these functions do exactly the same thing: they replace the process image with a process image from the absolute executable whose file name is specified in the first argument (`path` or `file`). They differ only in the manner in which they supply the parameters:

- The parameter `path` specifies an absolute pathname. If this file does not exist, the call fails.
- Alternatively, the parameter `file` specifies a file to be searched via the `PATH` environment variable, the way the shell does when a file name is specified.
- The parameter `argv` is a pointer to a `NULL` terminated list of parameters.
- Alternatively, you can place the arguments, including the terminating `NULL`, in the call as a series of `arg`s.
- If the parameter `envp` is specified, it is a pointer to a `NULL`-terminated list of environment variables. This is typically used when the child process should be given a different environment from the parent process.
- If `envp` is not specified, the environment variables are taken from the parent's environment (via the global pointer `environ`).

One further function deserves mention: `exect`, which is supplied only in newer BSD systems, takes the same parameters as `execve`, but enables program tracing facilities.

The total storage available for the argument list and the enviroment varies from system to system. System V traditionally has only 5120 characters. POSIX.1 requires at least 20480 characters, and this is the standard value for newer BSD systems. Many *Makefile*s take advantage of these large parameter lists, and frequently a package fails to build under System V because the parameter lists are too long: you get the message

```
make: execve: /bin/sh: Arg list too long
```

We looked at what we can do to solve these problems in Chapter 5, *Building the package*, page 74.

getrlimit and setrlimit

The Seventh Edition made a number of arbitrary choices about kernel limits. For example, each process was allowed to have 50 files open at any one time. In the course of time, a number of these kernel limits were made configurable, and some systems allowed the process to modify them directly, up to a "hard" limit. SunOS, BSD and System V.4 supply the system calls `getrlimit` and `setrlimit` in order to manipulate this configuration information:

```
#include <sys/time.h>
#include <sys/resource.h>
struct rlimit
```

```
    {
    int rlim_cur;                       /* current (soft) limit */
    int rlim_max;                       /* hard limit */
    };

int getrlimit (int resource, struct rlimit *rlp);
int setrlimit (int resource, struct rlimit *rlp);
```

The `rlimit` structure defines two values for each resource, the current value and the maximum value. `getrlimit` returns this information, `setrlimit` sets a new current value. Table 12-1 shows which limits can be set:

Table 12−1: `getrlimit` and `setrlimit` resources

resource	System	Description
`RLIMIT_CORE`	all	The maximum size, in bytes, of a core image file.
`RLIMIT_CPU`	all	The maximum amount of CPU time that a process may consume.
`RLIMIT_DATA`	all	The maximum size, in bytes, of the process data segment.
`RLIMIT_FSIZE`	all	The largest size, in bytes, that any file may attain.
`RLIMIT_MEMLOCK`	4.4BSD	The maximum size, in bytes, which a process may lock into memory using the `mlock` function.
`RLIMIT_NOFILE`	all	The maximum number of files that a process may open at one time. This is also one more than the highest file number that the process may use.
`RLIMIT_NPROC`	4.4BSD	The maximum number of simultaneous processes for the current user id.
`RLIMIT_RSS`	4.4BSD, SunOS 4	The maximum size, in bytes, that the resident set of a processes may attain. This limits the amount of physical memory that a process can occupy.
`RLIMIT_STACK`	all	The maximum size, in bytes, that the stack segment of a processes may attain.
`RLIMIT_VMEM`	System V.4	The maximum size, in bytes, that the mapped address space of a processes may attain.

If your system doesn't have these functions, there's not much you can do except guess. In some cases, header files will contain similar information declared as constants, but it's not a very satisfactory alternative.

Process groups

Where other operating systems use a single program to perform an operation, UNIX frequently uses a group of cooperating processes. It's useful to be able to define such a group, particularly when they access terminals. *Advanced Programming in the UNIX environment*, by Richard Stevens, describes all you will want to know about process groups. Here, we'll look at some minor differences in implementations.

setpgid

`setpgid` adds a process to a process group:

```
#include <unistd.h>

int setpgid (pid_t pid, pid_t pgrp);
```

`pid` is the process ID of the process that is to be added to the process group, and `pgrp` is the process group to which it should be added. It returns 0 on success and -1 with an error code in `errno` on failure.

Normally you will see `setpgid` used to add the calling process to a group; this can be done by setting `pid` to 0. System V versions also allow `pgrp` to be 0: this specifies that the process id should be the same as `pid`, and that this process will become a process group leader.

setpgrp

`setpgrp` is obsolescent. There are two different implementations, both of which duplicate functionality supplied by other functions:

- In more modern BSD systems, it is the same thing as `setpgid`:

  ```
  int setpgrp (pid_t pid, pid_t pgrp);    BSD versions
  ```

- In System V, it creates a new process group with the calling process as group leader, and adds the calling process to the group. It also releases the controlling terminal of the calling process. This is the same thing as `setsid`:

  ```
  int setpgrp ();                         System V versions
  ```

If you run into trouble with this function, it's best to replace it with `setpgid` or `setsid`, depending on the functionality that was intended.

setsid

`setsid` creates a new process group with the calling process as group leader, and adds the calling process to the group. It also releases the calling process from its controlling terminal:

```
#include <unistd.h>

int setsid ();
```

Real and effective user IDs

Occasionally the UNIX security system causes unintended problems: a trusted program may require access to facilities to which the user should not have unlimited access. For example, the program *ps* requires access to */dev/kmem*, kernel memory, which is normally accessible only to the super user. Serial communication programs such as *uucp* require access to the serial ports, but in order to avoid conflicts, only trusted users have access to the ports.

UNIX solves this problem by allowing the programs always to run as a specific user or group. If you execute a program that has the *setuid* bit set in the file permissions, it runs as if its owner had *exec*ed it, no matter who really started it. Similarly, the *setgid* bit causes the program to run as if it had been executed in the group to which the file belongs. These user and group ids are called *effective user ID* and *effective group ID*, and they are the only permissions that are relevant when a process accesses a file.

Similar considerations apply to group IDs. In the following discussion, we'll consider user IDs, but unless mentioned otherwise, everything I say about user IDs also applies to group IDs.

A number of subtle problems arise from this scheme. One of the most obvious ones is that programs frequently also need to be able to access your files. There's no guarantee that this will always work. For example, *uucp* needs to be setuid to user `uucp` in order to access the communication ports, but it also frequently needs to transfer data to your home directory. If your permissions are set so that *uucp* can't access your home directory, it will not be able to perform the transfer. This is obviously not the intention: somehow, *uucp* needs access both to the serial ports and to your files.

This means that we need to maintain at least two user IDs, the *effective user ID* and the *real user ID*. Modern systems also supply a *saved set user ID*. On System V.4, it's a configuration option (set the configuration constant `_POSIX_SAVED_IDS`). BSD uses the saved set user ID in a different way from System V, as we will see below.

The system manipulates user IDs in the following ways:

- If you execute a program that is not *setuid*, it sets all IDs to the effective user ID of the process that executes it.
- If you execute a program that has the *setuid* permission set, it sets the effective user ID to the owner of the program, and the real user ID to the effective ID of the process that executes it. If there is a saved set user ID, it also sets it to the owner of the program.
- At run time you can change between IDs with the system call `setuid`. There are also a number of alternative calls. We'll look at them in the following sections.

setuid

`setuid` changes the effective user ID. If your current effective user ID is *root*, you can set it to any valid user ID. There, unfortunately, the similarity ends:

- In systems without a saved set user ID, including SunOS 4 and System V.3, `setuid` sets the effective user ID and the real user ID if the current effective user ID is *root*, otherwise

it sets only the effective user ID. The function call succeeds if the argument to `setuid` is the real user ID or the effective user ID, or if the effective user ID is *root*. Once you have changed away from the old effective user ID and *root*, there is no way to change back.

- On System V systems with saved set user ID, `setuid` sets the effective user ID and the real user ID if the current effective user ID is *root*, otherwise it sets only the effective user ID. It does not change the saved set user ID. The function call succeeds if the argument to `setuid` is the real user ID, the effective user ID, or the saved set user ID, or if the effective user ID is *root*. This means that you can switch back and forth between the ID of the program owner and the ID of the process which started it.
- On BSD systems with saved set user ID, `setuid` sets the real, effective, and saved set user IDs. The function call succeeds if the argument to `setuid` is the real user ID, or if the effective user ID is *root*. Unlike System V.4, non-*root* users cannot use `setuid` to set the user ID to the saved set user ID. The saved set user ID is of no use to BSD `setuid`—instead, BSD systems use `seteuid`, which sets only the effective user ID to either the real user ID or the saved set user ID.

setreuid

BSD versions since 4.2BSD have the system call `setreuid`, which takes two parameters:

```
int setreuid (int ruid, int euid);
```

You can use it to swap the effective and real user IDs, so you don't really need a saved set user ID. For non-privileged users, `ruid` and `euid` can be either the current real user ID or the current effective user ID, or `-1` to indicate no change. This function was needed in BSD up to and including 4.3BSD, since these versions did not support the concept of a saved set user ID. On non-BSD systems only, you can replace this function with `setuid` if your system supports saved set user IDs.

seteuid

As we noted above, BSD `setuid` cannot change to the saved set user ID. The BSD solution to this problem, which has been proposed for adoption in a new revision of POSIX.1, is the function `seteuid`. It sets the effective user ID to `euid` if `euid` corresponds either to the real user ID or the saved set user ID. Unlike `setuid`, it sets only the effective user ID.

setruid

In addition to `seteuid`, BSD systems provide the call `setruid`, which sets the real user ID to the effective or real user ID. `setruid` is considered non-portable. Future BSD releases plan to drop it.

Comparison of user ID calls

User IDs are much more complicated than they should be. In fact, there are only two things you'll want to do, and for non-root users they work only with programs which have setuid permissions: change from the initial effective user ID to the real user ID, and change back again. Changing from effective to real user ID is simple: in all systems, you can use the `setuid` system call, though in 4.3BSD and SunOS 4 this will mean that you can't change back. In these systems, it's better to use code like

```
int euid = geteuid ();          /* get current effective user ID */
int ruid = getuid ();           /* and real user ID */
setreuid (euid, ruid);          /* and swap them */
```

Changing back again is more complicated:

- On older systems, including XENIX and System V.3, and on System V.4 systems without `_POSIX_SAVED_IDS`, you can't do it. For the older systems, about the only workaround is not to change away from the initial effective user ID—you might be able to spawn a process which does the necessary work under the real user ID.
- On BSD systems up to and including 4.3BSD, and under SunOS 4, you can do it only if you changed with `setreuid`, as in the example above. In this case, you just need to continue with

  ```
  setreuid (ruid, euid);
  ```

- On System V.4 systems with `_POSIX_SAVED_IDS`, use `setuid (ssuid)`, where `ssuid` is the saved set user ID. You can get the value of `ssuid` by calling `geteuid` before changing the initial effective user ID, since they're the same at program start.
- On BSD systems which support saved set user IDs, use `seteuid (ssuid)`. As with System V.4, you can get the value of `ssuid` by calling `geteuid` before changing the initial effective user ID.

vfork

`vfork` was introduced in 3BSD as a more efficient version of `fork`: in those days, `fork` copied each data area page of the parent process for the child process, which could take a considerable time. Typically, the first thing a child does is to call `exec` to run a new program, which discards the data pages, so this was effectively wasted time. `vfork` modified this behaviour so that the pages were shared and not copied.

This is inherently very dangerous: very frequently the parent waits until the child has done something before continuing. During this time, the child can modify the parent's data, since it is shared. More modern techniques, such as *copy on write*[*], have eliminated the need for this function. You *should* be able to replace it with `fork` (the semantics are identical). Unfortunately, some obscene programs rely on the fact that they can manipulate the parent's data

* With copy on write, the data pages are set to be write-protected. The first write causes an interrupt, effectively a bus error, which the system intercepts. The system makes a copy of the single page and resets write protection for both the original and the copy, allowing the write to proceed.

before the parent continues. These programs need to be fixed.

wait and friends

`wait` has been in UNIX as long as anybody can remember:

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

pid_t wait (int *status);
```

Unfortunately, various flavours define the value of the status return differently. This is a cosmetic difference, not a real difference: the status information consists of a number of bit fields that depend on the kind of status:

- The low-order 7 bits contain the number of the signal that terminated the process, or 0 if the process called `exit`.
- The bit `0x80` is set if a core dump was taken.
- The next 8 bits are the return code if the process called `exit`.

If the process is stopped (if it can be restarted), the low-order 8 bits are set to `127` (`0x7f`), and the next byte contains the number of the signal that stopped the process.

This information is the same on all versions of UNIX, but there is no agreement on how to represent this information. Older BSD systems defined a union to represent it:

```
union __wait
  {
  int w_status;                         /* status as int */
  struct
    {
    unsigned short w_Termsig:7;         /* termination signal */
    unsigned short w_Coredump:1;        /* core dump indicator */
    unsigned short w_Retcode:8;         /* exit code if w_termsig==0 */
    }
  w_T;
  struct
    {
    unsigned short w_Stopval:8;         /* == W_STOPPED if stopped */
    unsigned short w_Stopsig:8;         /* signal that stopped us */
    }
   w_S;
};
```

Modern systems define macros:

- `WIFEXITED (status)` is true if the process terminated via a call to `exit`. If this is true, `WEXITSTATUS (status)` returns the low order 8 bits of the process' exit value.
- `WIFSIGNALED (status)` is true if the process was terminated by receiving a signal. If this is true, the following macros apply:

  - `WTERMSIG (status)` evaluates to the number of the signal that caused the termination of the process.
  - `WCOREDUMP (status)` is true if a core dump was created.

- `WIFSTOPPED (status)` is true if the process is stopped and can be restarted. This macro can be true only if the `waitpid` call specified the `WUNTRACED` option or if the child process is being traced. If this is true, `WSTOPSIG (status)` returns the number of the signal that caused the process to stop.

Some systems offer both of these options, sometimes incompletely. For example, SunOS 4 defines `w_Coredump` in the `union __wait`, but does not define the corresponding `WCOREDUMP` macro.

These varying differences cause problems out of all proportion to the importance of the information contained. In particular, the newer macros do not allow you to change the status, you can only read it. Some programs, for example BSD *make*, modify the status. This makes it difficult to port it to System V or another system which does not understand `union wait`.

waitpid

`waitpid` is a variant of `wait` that waits for a specific process to terminate. It is part of all modern UNIX implementations:

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

pid_t waitpid (pid_t wpid, int *status, int options);
```

`waitpid` waits for process `pid` to terminate. Its behaviour is governed by a number of bitmapped options:

- Set `WNOHANG` to specify to return immediately, even if no status is available. If the status is not available, the functions return the process number 0. Not all systems support this behaviour.
- Specify `WUNTRACED` if you want the status of stopped processes as well as processes that have terminated. Some systems do not return complete status information for stopped processes.
- Under System V.4, use `WCONTINUED` to report the status of any process that has continued (in other words, one that is no longer stopped) since the last status report.
- Also under System V.4 you can set the option `WNOWAIT` to specify that the process should not terminate (it remains a zombie). This means that you can call `waitpid` again and get the same information.

The value of `status` is the same as with `wait`—see the previous section for further details.

If you run into problems with `waitpid`, it may be a bug: some versions of System V.3, including most current versions of SCO UNIX, return a process ID if a process is waiting, and an error number such as `ECHILD` (10) if nothing is waiting, so if your freshly ported program keeps reporting the demise of process 10, this could be the problem. It's almost impossible to

work around this bug—about the only thing you can do is to use some other system call.

wait3 and wait4

Newer BSD implementations supply the functions `wait3` and `wait4` as alternatives to `wait`. They correspond to `wait` and `waitpid` respectively, but return an additional parameter `rusage` with accounting information for the terminated process:

```
pid_t wait3 (int *status, int options, struct rusage *rusage);
pid_t wait4 (pid_t wpid, int *status, int options, struct rusage *rusage);
```

Not all implementations return usage information to `rusage` when the process is stopped (and not terminated). The definition of `struct rusage` is implementation-dependent and defined in *sys/resource.h*. See the file *sys/sys/resource.h* in the 4.4BSD Lite distribution for further details.