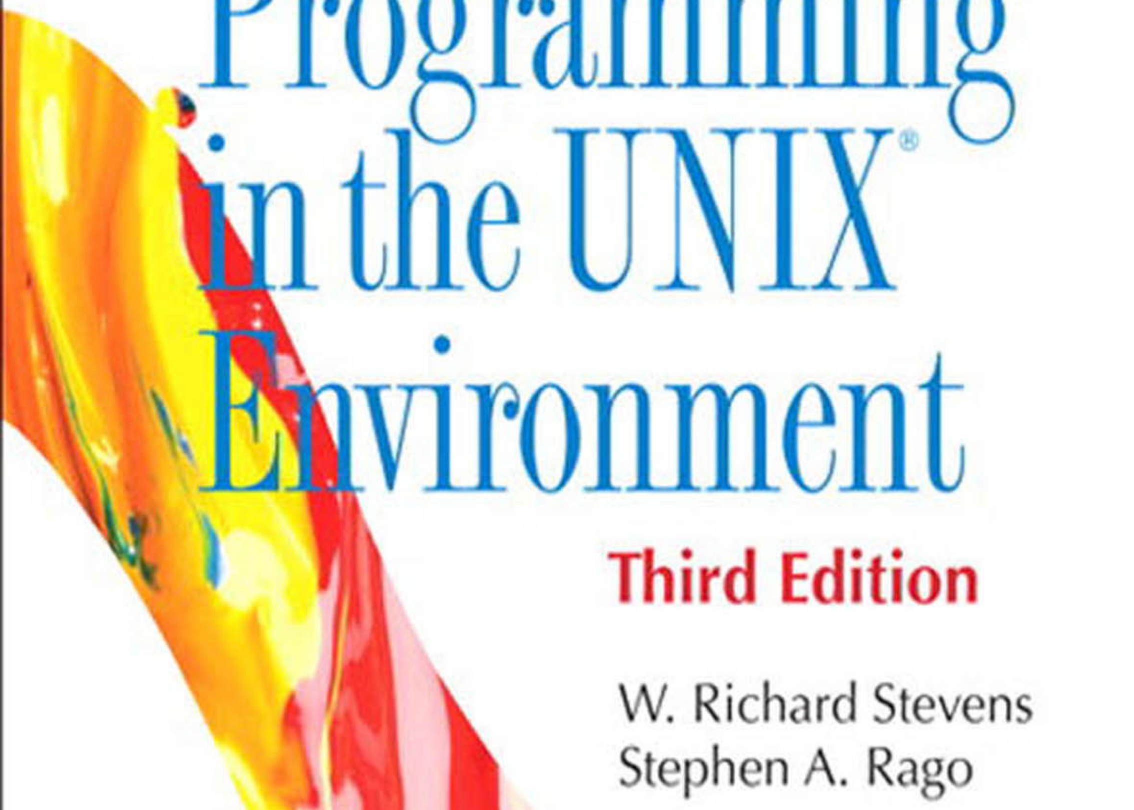
Advanced Programming in the UNIX® Environment

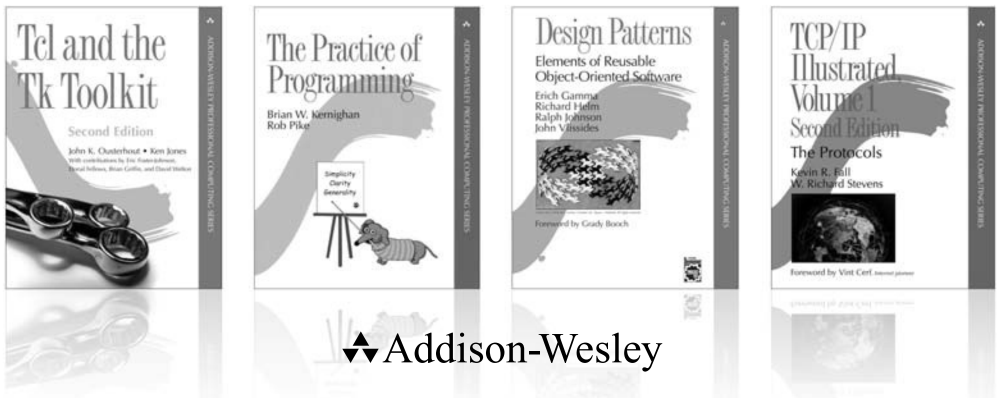
Third Edition



W. Richard Stevens
Stephen A. Rago

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

The Addison-Wesley Professional Computing Series



Visit informit.com/series/professionalcomputing
for a complete list of available publications.

The Addison-Wesley Professional Computing Series was created in 1990 to provide serious programmers and networking professionals with well-written and practical reference books. There are few places to turn for accurate and authoritative books on current and cutting-edge technology. We hope that our books will help you understand the state of the art in programming languages, operating systems, and networks.

Consulting Editor Brian W. Kernighan

Make sure to connect with us!
informit.com/socialconnect

Addison Wesley | informIT.com THE TRUSTED TECHNOLOGY LEARNING SOURCE | Safari Books Online

ALWAYS LEARNING PEARSON

Given the login name, we can then use it to look up the user in the password file—to determine the login shell, for example—using getpwnam.

> To find the login name, UNIX systems have historically called the ttyname function (Section 18.9) and then tried to find a matching entry in the utmp file (Section 6.8). FreeBSD and Mac OS X store the login name in the session structure associated with the process table entry and provide system calls to fetch and store this name.
>
> System V provided the cuserid function to return the login name. This function called getlogin and, if that failed, did a getpwuid(getuid()). The IEEE Standard 1003.1-1988 specified cuserid, but it called for the effective user ID to be used, instead of the real user ID. The 1990 version of POSIX.1 dropped the cuserid function.
>
> The environment variable LOGNAME is usually initialized with the user's login name by login(1) and inherited by the login shell. Realize, however, that a user can modify an environment variable, so we shouldn't use LOGNAME to validate the user in any way. Instead, we should use getlogin.

8.16 Process Scheduling

Historically, the UNIX System provided processes with only coarse control over their scheduling priority. The scheduling policy and priority were determined by the kernel. A process could choose to run with lower priority by adjusting its *nice value* (thus a process could be "nice" and reduce its share of the CPU by adjusting its nice value). Only a privileged process was allowed to increase its scheduling priority.

The real-time extensions in POSIX added interfaces to select among multiple scheduling classes and fine-tune their behavior. We discuss only the interfaces used to adjust the nice value here; they are part of the XSI option in POSIX.1. Refer to Gallmeister [1995] for more information on the real-time scheduling extensions.

In the Single UNIX Specification, nice values range from 0 to (2*NZERO)−1, although some implementations support a range from 0 to 2*NZERO. Lower nice values have higher scheduling priority. Although this might seem backward, it actually makes sense: the more nice you are, the lower your scheduling priority is. NZERO is the default nice value of the system.

> Be aware that the header file defining NZERO differs among systems. In addition to the header file, Linux 3.2.0 makes the value of NZERO accessible through a nonstandard sysconf argument (_SC_NZERO).

A process can retrieve and change its nice value with the nice function. With this function, a process can affect only its own nice value; it can't affect the nice value of any other process.

```
#include <unistd.h>

int nice(int incr);
```

Returns: new nice value − NZERO if OK, −1 on error

The *incr* argument is added to the nice value of the calling process. If *incr* is too large, the system silently reduces it to the maximum legal value. Similarly, if *incr* is too small, the system silently increases it to the minimum legal value. Because −1 is a legal successful return value, we need to clear `errno` before calling `nice` and check its value if `nice` returns −1. If the call to `nice` succeeds and the return value is −1, then `errno` will still be zero. If `errno` is nonzero, it means that the call to `nice` failed.

The `getpriority` function can be used to get the nice value for a process, just like the `nice` function. However, `getpriority` can also get the nice value for a group of related processes.

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

int getpriority(int which, id_t who);
```

Returns: nice value between −`NZERO` and `NZERO−1` if OK, −1 on error

The *which* argument can take on one of three values: `PRIO_PROCESS` to indicate a process, `PRIO_PGRP` to indicate a process group, and `PRIO_USER` to indicate a user ID. The *which* argument controls how the *who* argument is interpreted and the *who* argument selects the process or processes of interest. If the *who* argument is 0, then it indicates the calling process, process group, or user (depending on the value of the *which* argument). When *which* is set to `PRIO_USER` and *who* is 0, the real user ID of the calling process is used. When the *which* argument applies to more than one process, the highest priority (lowest value) of all the applicable processes is returned.

The `setpriority` function can be used to set the priority of a process, a process group, or all the processes belonging to a particular user ID.

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

int setpriority(int which, id_t who, int value);
```

Returns: 0 if OK, −1 on error

The *which* and *who* arguments are the same as in the `getpriority` function. The *value* is added to `NZERO` and this becomes the new nice value.

> The `nice` system call originated with an early PDP-11 version of the Research UNIX System. The `getpriority` and `setpriority` functions originated with 4.2BSD.

The Single UNIX Specification leaves it up to the implementation whether the nice value is inherited by a child process after a `fork`. However, XSI-compliant systems are required to preserve the nice value across a call to `exec`.

> A child process inherits the nice value from its parent process in FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, and Solaris 10.

Example

The program in Figure 8.30 measures the effect of adjusting the nice value of a process. Two processes run in parallel, each incrementing its own counter. The parent runs with the default nice value, and the child runs with an adjusted nice value as specified by the optional command argument. After running for 10 seconds, both processes print the value of their counter and exit. By comparing the counter values for different nice values, we can get an idea how the nice value affects process scheduling.

```
#include "apue.h"
#include <errno.h>
#include <sys/time.h>

#if defined(MACOS)
#include <sys/syslimits.h>
#elif defined(SOLARIS)
#include <limits.h>
#elif defined(BSD)
#include <sys/param.h>
#endif

unsigned long long count;
struct timeval end;

void
checktime(char *str)
{
    struct timeval  tv;

    gettimeofday(&tv, NULL);
    if (tv.tv_sec >= end.tv_sec && tv.tv_usec >= end.tv_usec) {
        printf("%s count = %lld\n", str, count);
        exit(0);
    }
}

int
main(int argc, char *argv[])
{
    pid_t   pid;
    char    *s;
    int     nzero, ret;
    int     adj = 0;

    setbuf(stdout, NULL);
#if defined(NZERO)
    nzero = NZERO;
#elif defined(_SC_NZERO)
    nzero = sysconf(_SC_NZERO);
#else
#error NZERO undefined
#endif
    printf("NZERO = %d\n", nzero);
    if (argc == 2)
        adj = strtol(argv[1], NULL, 10);
    gettimeofday(&end, NULL);
    end.tv_sec += 10;   /* run for 10 seconds */

    if ((pid = fork()) < 0) {
```

```
        err_sys("fork failed");
    } else if (pid == 0) {  /* child */
        s = "child";
        printf("current nice value in child is %d, adjusting by %d\n",
          nice(0)+nzero, adj);
        errno = 0;
        if ((ret = nice(adj)) == -1 && errno != 0)
            err_sys("child set scheduling priority");
        printf("now child nice value is %d\n", ret+nzero);
    } else {        /* parent */
        s = "parent";
        printf("current nice value in parent is %d\n", nice(0)+nzero);
    }
    for(;;) {
        if (++count == 0)
            err_quit("%s counter wrap", s);
        checktime(s);
    }
}
```

Figure 8.30 Evaluate the effect of changing the nice value

We run the program twice: once with the default nice value, and once with the highest valid nice value (the lowest scheduling priority). We run this on a uniprocessor Linux system to show how the scheduler shares the CPU among processes with different nice values. With an otherwise idle system, a multiprocessor system (or a multicore CPU) would allow both processes to run without the need to share a CPU, and we wouldn't see much difference between two processes with different nice values.

```
$ ./a.out
NZERO = 20
current nice value in parent is 20
current nice value in child is 20, adjusting by 0
now child nice value is 20
child count = 1859362
parent count = 1845338
$ ./a.out 20
NZERO = 20
current nice value in parent is 20
current nice value in child is 20, adjusting by 20
now child nice value is 39
parent count = 3595709
child count = 52111
```

When both processes have the same nice value, the parent process gets 50.2% of the CPU and the child gets 49.8% of the CPU. Note that the two processes are effectively treated equally. The percentages aren't exactly equal, because process scheduling isn't exact, and because the child and parent perform different amounts of processing between the time that the end time is calculated and the time that the processing loop begins.

In contrast, when the child has the highest possible nice value (the lowest priority), we see that the parent gets 98.5% of the CPU, while the child gets only 1.5% of the CPU. These values will vary based on how the process scheduler uses the nice value, so a different UNIX system will produce different ratios. □

8.17 Process Times

In Section 1.10, we described three times that we can measure: wall clock time, user CPU time, and system CPU time. Any process can call the `times` function to obtain these values for itself and any terminated children.

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

clock_t times(struct tms *buf);
```

Returns: elapsed wall clock time in clock ticks if OK, –1 on error

This function fills in the `tms` structure pointed to by *buf*:

```
struct tms {
  clock_t  tms_utime;  /* user CPU time */
  clock_t  tms_stime;  /* system CPU time */
  clock_t  tms_cutime; /* user CPU time, terminated children */
  clock_t  tms_cstime; /* system CPU time, terminated children */
};
```

Note that the structure does not contain any measurement for the wall clock time. Instead, the function returns the wall clock time as the value of the function, each time it's called. This value is measured from some arbitrary point in the past, so we can't use its absolute value; instead, we use its relative value. For example, we call `times` and save the return value. At some later time, we call `times` again and subtract the earlier return value from the new return value. The difference is the wall clock time. (It is possible, though unlikely, for a long-running process to overflow the wall clock time; see Exercise 1.5.)

The two structure fields for child processes contain values only for children that we have waited for with one of the `wait` functions discussed earlier in this chapter.

All the `clock_t` values returned by this function are converted to seconds using the number of clock ticks per second—the `_SC_CLK_TCK` value returned by `sysconf` (Section 2.5.4).

> Most implementations provide the `getrusage`(2) function. This function returns the CPU times and 14 other values indicating resource usage. Historically, this function originated with the BSD operating system, so BSD-derived implementations generally support more of the fields than do other implementations.

Example

The program in Figure 8.31 executes each command-line argument as a shell command string, timing the command and printing the values from the `tms` structure.

```
#include "apue.h"
#include <sys/times.h>

static void pr_times(clock_t, struct tms *, struct tms *);
static void do_cmd(char *);

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

    setbuf(stdout, NULL);
    for (i = 1; i < argc; i++)
        do_cmd(argv[i]);    /* once for each command-line arg */
    exit(0);
}

static void
do_cmd(char *cmd)       /* execute and time the "cmd" */
{
    struct tms  tmsstart, tmsend;
    clock_t     start, end;
    int         status;

    printf("\ncommand: %s\n", cmd);

    if ((start = times(&tmsstart)) == -1)   /* starting values */
        err_sys("times error");

    if ((status = system(cmd)) < 0)         /* execute command */
        err_sys("system() error");

    if ((end = times(&tmsend)) == -1)       /* ending values */
        err_sys("times error");

    pr_times(end-start, &tmsstart, &tmsend);
    pr_exit(status);
}

static void
pr_times(clock_t real, struct tms *tmsstart, struct tms *tmsend)
{
    static long     clktck = 0;

    if (clktck == 0)    /* fetch clock ticks per second first time */
        if ((clktck = sysconf(_SC_CLK_TCK)) < 0)
            err_sys("sysconf error");

    printf("  real:  %7.2f\n", real / (double) clktck);
    printf("  user:  %7.2f\n",
      (tmsend->tms_utime - tmsstart->tms_utime) / (double) clktck);
    printf("  sys:   %7.2f\n",
      (tmsend->tms_stime - tmsstart->tms_stime) / (double) clktck);
    printf("  child user:  %7.2f\n",
```