28

PROCESS CREATION AND PROGRAM EXECUTION IN MORE DETAIL

This chapter extends the material presented in Chapters 24 to 27 by covering a variety of topics related to process creation and program execution. We describe process accounting, a kernel feature that writes an accounting record for each process on the system as it terminates. We then look at the Linux-specific *clone()* system call, which is the low-level API that is used to create threads on Linux. We follow this with some comparisons of the performance of *fork()*, *vfork()*, and *clone()*. We conclude with a summary of the effects of *fork()* and *exec()* on the attributes of a process.

28.1 Process Accounting

When process accounting is enabled, the kernel writes an accounting record to the system-wide process accounting file as each process terminates. This accounting record contains various information maintained by the kernel about the process, including its termination status and how much CPU time it consumed. The accounting file can be analyzed by standard tools (*sa(8)* summarizes information from the accounting file, and *lastcomm(1)* lists information about previously executed commands) or by tailored applications.

In kernels before 2.6.10, a separate process accounting record was written for each thread created using the NPTL threading implementation. Since kernel 2.6.10, a single accounting record is written for the entire process when the last thread terminates. Under the older LinuxThreads threading implementation, a single process accounting record is always written for each thread.

Historically, the primary use of process accounting was to charge users for consumption of system resources on multiuser UNIX systems. However, process accounting can also be useful for obtaining information about a process that was not otherwise monitored and reported on by its parent.

Although available on most UNIX implementations, process accounting is not specified in SUSv3. The format of the accounting records, as well as the location of the accounting file, vary somewhat across implementations. We describe the details for Linux in this section, noting some variations from other UNIX implementations along the way.

On Linux, process accounting is an optional kernel component that is configured via the option `CONFIG_BSD_PROCESS_ACCT`.

Enabling and disabling process accounting

The *acct()* system call is used by a privileged (`CAP_SYS_PACCT`) process to enable and disable process accounting. This system call is rarely used in application programs. Normally, process accounting is enabled at each system restart by placing appropriate commands in the system boot scripts.

```
#define _BSD_SOURCE
#include <unistd.h>

int acct(const char *acctfile);
```

Returns 0 on success, or –1 on error

To enable process accounting, we supply the pathname of an *existing* regular file in *acctfile*. A typical pathname for the accounting file is `/var/log/pacct` or `/usr/account/pacct`. To disable process accounting, we specify *acctfile* as `NULL`.

The program in Listing 28-1 uses *acct()* to switch process accounting on and off. The functionality of this program is similar to the shell *accton(8)* command.

Listing 28-1: Turning process accounting on and off

procexec/acct_on.c

```
#define _BSD_SOURCE
#include <unistd.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    if (argc > 2 || (argc > 1 && strcmp(argv[1], "--help") == 0))
        usageErr("%s [file]\n");
    if (acct(argv[1]) == -1)
        errExit("acct");

    printf("Process accounting %s\n",
            (argv[1] == NULL) ? "disabled" : "enabled");
    exit(EXIT_SUCCESS);
}
```

procexec/acct_on.c

Process accounting records

Once process accounting is enabled, an *acct* record is written to the accounting file as each process terminates. The *acct* structure is defined in `<sys/acct.h>` as follows:

```
typedef u_int16_t comp_t;  /* See text */

struct acct {
    char      ac_flag;     /* Accounting flags (see text) */
    u_int16_t ac_uid;      /* User ID of process */
    u_int16_t ac_gid;      /* Group ID of process */
    u_int16_t ac_tty;      /* Controlling terminal for process (may be
                              0 if none, e.g., for a daemon) */
    u_int32_t ac_btime;    /* Start time (time_t; seconds since the Epoch) */
    comp_t    ac_utime;    /* User CPU time (clock ticks) */
    comp_t    ac_stime;    /* System CPU time (clock ticks) */
    comp_t    ac_etime;    /* Elapsed (real) time (clock ticks) */
    comp_t    ac_mem;      /* Average memory usage (kilobytes) */
    comp_t    ac_io;       /* Bytes transferred by read(2) and write(2)
                              (unused) */
    comp_t    ac_rw;       /* Blocks read/written (unused) */
    comp_t    ac_minflt;   /* Minor page faults (Linux-specific) */
    comp_t    ac_majflt;   /* Major page faults (Linux-specific) */
    comp_t    ac_swaps;    /* Number of swaps (unused; Linux-specific) */
    u_int32_t ac_exitcode; /* Process termination status */
#define ACCT_COMM 16
    char      ac_comm[ACCT_COMM+1];
                           /* (Null-terminated) command name
                              (basename of last execed file) */
    char      ac_pad[10];  /* Padding (reserved for future use) */
};
```

Note the following points regarding the *acct* structure:

- The *u_int16_t* and *u_int32_t* data types are 16-bit and 32-bit unsigned integers.
- The *ac_flag* field is a bit mask recording various events for the process. The bits that can appear in this field are shown in Table 28-1. As indicated in the table, some of these bits are not present on all UNIX implementations. A few other implementations provide additional bits in this field.
- The *ac_comm* field records the name of the last command (program file) executed by this process. The kernel records this value on each *execve()*. On some other UNIX implementations, this field is limited to 8 characters.

- The *comp_t* type is a kind of floating-point number. Values of this type are sometimes called *compressed clock ticks*. The floating-point value consists of a 3-bit, base-8 exponent, followed by a 13-bit mantissa; the exponent can represent a factor in the range 8^0=1 to 8^7 (2,097,152). For example, a mantissa of 125 and an exponent of 1 represent the value 1000. Listing 28-2 defines a function (*comptToLL()*) to convert this type to *long long*. We need to use the type *long long* because the 32 bits used to represent an *unsigned long* on x86-32 are insufficient to hold the largest value that can be represented in *comp_t*, which is $(2^{13} - 1) * 8^7$.
- The three time fields defined with the type *comp_t* represent time in system clock ticks. Therefore, we must divide these times by the value returned by *sysconf(_SC_CLK_TCK)* in order to convert them to seconds.
- The *ac_exitcode* field holds the termination status of the process (described in Section 26.1.3). Most other UNIX implementations instead provide a single-byte field named *ac_stat*, which records only the signal that killed the process (if it was killed by a signal) and a bit indicating whether that signal caused the process to dump core. BSD-derived implementations don't provide either field.

Table 28-1: Bit values for the *ac_flag* field of process accounting records

Bit	Description
`AFORK`	Process was created by *fork()*, but did not *exec()* before terminating
`ASU`	Process made use of superuser privileges
`AXSIG`	Process was terminated by a signal (not present on some implementations)
`ACORE`	Process produced a core dump (not present on some implementations)

Because accounting records are written only as processes terminate, they are ordered by termination time (a value not recorded in the record), rather than by process start time (*ac_btime*).

> If the system crashes, no accounting record is written for any processes that are still executing.

Since writing records to the accounting file can rapidly consume disk space, Linux provides the `/proc/sys/kernel/acct` virtual file for controlling the operation of process accounting. This file contains three numbers, defining (in order) the parameters *high-water*, *low-water*, and *frequency*. Typical defaults for these three parameters are 4, 2, and 30. If process accounting is enabled and the amount of free disk space falls below *low-water* percent, accounting is suspended. If the amount of free disk space later rises above *high-water* percent, then accounting is resumed. The *frequency* value specifies how often, in seconds, checks should be made on the percentage of free disk space.

Example program

The program in Listing 28-2 displays selected fields from the records in a process accounting file. The following shell session demonstrates the use of this program. We begin by creating a new, empty process accounting file and enabling process accounting:

```
$ su                            Need privilege to enable process accounting
Password:
# touch pacct
# ./acct_on pacct               This process will be first entry in accounting file
Process accounting enabled
# exit                          Cease being superuser
```

At this point, three processes have already terminated since we enabled process accounting. These processes executed the *acct_on*, *su*, and *bash* programs. The *bash* process was started by *su* to run the privileged shell session.

Now we run a series of commands to add further records to the accounting file:

```
$ sleep 15 &
[1] 18063
$ ulimit -c unlimited           Allow core dumps (shell built-in)
$ cat                           Create a process
Type Control-\ (generates SIGQUIT, signal 3) to kill cat process
Quit (core dumped)
$
Press Enter to see shell notification of completion of sleep before next shell prompt
[1]+  Done          sleep 15
$ grep xxx badfile              grep fails with status of 2
grep: badfile: No such file or directory
$ echo $?                       The shell obtained status of grep (shell built-in)
2
```

The next two commands run programs that we presented in previous chapters (Listing 27-1, on page 566, and Listing 24-1, on page 517). The first command runs a program that execs the file `/bin/echo`; this results in an accounting record with the command name *echo*. The second command creates a child process that doesn't perform an *exec()*.

```
$ ./t_execve /bin/echo
hello world goodbye
$ ./t_fork
PID=18350 (child) idata=333 istack=666
PID=18349 (parent) idata=111 istack=222
```

Finally, we use the program in Listing 28-2 to view the contents of the accounting file:

```
$ ./acct_view pacct
command  flags   term.  user     start time           CPU   elapsed
                 status                               time    time
acct_on   -S--       0  root     2010-07-23 17:19:05  0.00    0.00
bash      ----       0  root     2010-07-23 17:18:55  0.02   21.10
su        -S--       0  root     2010-07-23 17:18:51  0.01   24.94
cat       --XC    0x83  mtk      2010-07-23 17:19:55  0.00    1.72
sleep     ----       0  mtk      2010-07-23 17:19:42  0.00   15.01
grep      ----   0x200  mtk      2010-07-23 17:20:12  0.00    0.00
echo      ----       0  mtk      2010-07-23 17:21:15  0.01    0.01
t_fork    F---       0  mtk      2010-07-23 17:21:36  0.00    0.00
t_fork    ----       0  mtk      2010-07-23 17:21:36  0.00    3.01
```

In the output, we see one line for each process that was created in the shell session. The *ulimit* and *echo* commands are shell built-in commands, so they don't result in the creation of new processes. Note that the entry for *sleep* appeared in the accounting file after the *cat* entry because the *sleep* command terminated after the *cat* command.

Most of the output is self-explanatory. The *flags* column shows single letters indicating which of the *ac_flag* bits is set in each record (see Table 28-1). Section 26.1.3 describes how to interpret the termination status values shown in the *term. status* column.

Listing 28-2: Displaying data from a process accounting file

procexec/acct_view.c

```
#include <fcntl.h>
#include <time.h>
#include <sys/stat.h>
#include <sys/acct.h>
#include <limits.h>
#include "ugid_functions.h"             /* Declaration of userNameFromId() */
#include "tlpi_hdr.h"

#define TIME_BUF_SIZE 100

static long long                /* Convert comp_t value into long long */
comptToLL(comp_t ct)
{
    const int EXP_SIZE = 3;             /* 3-bit, base-8 exponent */
    const int MANTISSA_SIZE = 13;       /* Followed by 13-bit mantissa */
    const int MANTISSA_MASK = (1 << MANTISSA_SIZE) - 1;
    long long mantissa, exp;

    mantissa = ct & MANTISSA_MASK;
    exp = (ct >> MANTISSA_SIZE) & ((1 << EXP_SIZE) - 1);
    return mantissa << (exp * 3);       /* Power of 8 = left shift 3 bits */
}

int
main(int argc, char *argv[])
{
    int acctFile;
    struct acct ac;
    ssize_t numRead;
    char *s;
    char timeBuf[TIME_BUF_SIZE];
    struct tm *loc;
    time_t t;

    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s file\n", argv[0]);

    acctFile = open(argv[1], O_RDONLY);
    if (acctFile == -1)
        errExit("open");
    printf("command  flags   term.  user     "
            "start time            CPU   elapsed\n");
    printf("                status         "
            "                     time    time\n");

    while ((numRead = read(acctFile, &ac, sizeof(struct acct))) > 0) {
        if (numRead != sizeof(struct acct))
            fatal("partial read");

        printf("%-8.8s  ", ac.ac_comm);

        printf("%c", (ac.ac_flag & AFORK) ? 'F' : '-') ;
        printf("%c", (ac.ac_flag & ASU)   ? 'S' : '-') ;
        printf("%c", (ac.ac_flag & AXSIG) ? 'X' : '-') ;
        printf("%c", (ac.ac_flag & ACORE) ? 'C' : '-') ;

#ifdef __linux__
        printf(" %#6lx   ", (unsigned long) ac.ac_exitcode);
#else   /* Many other implementations provide ac_stat instead */
        printf(" %#6lx   ", (unsigned long) ac.ac_stat);
#endif

        s = userNameFromId(ac.ac_uid);
        printf("%-8.8s ", (s == NULL) ? "???" : s);

        t = ac.ac_btime;
        loc = localtime(&t);
        if (loc == NULL) {
            printf("???Unknown time???  ");
        } else {
            strftime(timeBuf, TIME_BUF_SIZE, "%Y-%m-%d %T ", loc);
            printf("%s ", timeBuf);
        }

        printf("%5.2f %7.2f ", (double) (comptToLL(ac.ac_utime) +
                    comptToLL(ac.ac_stime)) / sysconf(_SC_CLK_TCK),
                (double) comptToLL(ac.ac_etime) / sysconf(_SC_CLK_TCK));
        printf("\n");
    }

    if (numRead == -1)
        errExit("read");

    exit(EXIT_SUCCESS);
}
```
procexec/acct_view.c

Process accounting Version 3 file format

Starting with kernel 2.6.8, Linux introduced an optional alternative version of the process accounting file that addresses some limitations of the traditional accounting file. To use this alternative version, known as *Version 3*, the `CONFIG_BSD_PROCESS_ACCT_V3` kernel configuration option must be enabled before building the kernel.

When using the Version 3 option, the only difference in the operation of process accounting is in the format of records written to the accounting file. The new format is defined as follows:

```
struct acct_v3 {
    char      ac_flag;        /* Accounting flags */
    char      ac_version;     /* Accounting version (3) */
    u_int16_t ac_tty;         /* Controlling terminal for process */
    u_int32_t ac_exitcode;    /* Process termination status */
    u_int32_t ac_uid;         /* 32-bit user ID of process */
    u_int32_t ac_gid;         /* 32-bit group ID of process */
    u_int32_t ac_pid;         /* Process ID */
    u_int32_t ac_ppid;        /* Parent process ID */
    u_int32_t ac_btime;       /* Start time (time_t) */
    float     ac_etime;       /* Elapsed (real) time (clock ticks) */
    comp_t    ac_utime;       /* User CPU time (clock ticks) */
    comp_t    ac_stime;       /* System CPU time (clock ticks) */
    comp_t    ac_mem;         /* Average memory usage (kilobytes) */
    comp_t    ac_io;          /* Bytes read/written (unused) */
    comp_t    ac_rw;          /* Blocks read/written (unused) */
    comp_t    ac_minflt;      /* Minor page faults */
    comp_t    ac_majflt;      /* Major page faults */
    comp_t    ac_swaps;       /* Number of swaps (unused; Linux-specific) */
#define ACCT_COMM 16
    char      ac_comm[ACCT_COMM];   /* Command name */
};
```

The following are the main differences between the *acct_v3* structure and the traditional Linux *acct* structure:

- The *ac_version* field is added. This field contains the version number of this type of accounting record. This field is always 3 for an *acct_v3* record.
- The fields *ac_pid* and *ac_ppid*, containing the process ID and parent process ID of the terminated process, are added.
- The *ac_uid* and *ac_gid* fields are widened from 16 to 32 bits, to accommodate the 32-bit user and group IDs that were introduced in Linux 2.4. (Large user and group IDs can't be correctly represented in the traditional *acct* file.)
- The type of the *ac_etime* field is changed from *comp_t* to *float*, to allow longer elapsed times to be recorded.

> We provide a Version 3 analog of the program in Listing 28-2 in the file `procexec/acct_v3_view.c` in the source code distribution for this book.

28.2 The *clone()* System Call

Like *fork()* and *vfork()*, the Linux-specific *clone()* system call creates a new process. It differs from the other two calls in allowing finer control over the steps that occur during process creation. The main use of *clone()* is in the implementation of threading libraries. Because *clone()* is not portable, its direct use in application programs should normally be avoided. We describe it here because it is useful background for the discussion of POSIX threads in Chapters 29 to 33, and also because it further illuminates the operation of *fork()* and *vfork()*.

```
#define _GNU_SOURCE
#include <sched.h>

int clone(int (*func) (void *), void *child_stack, int flags, void *func_arg, ...
          /* pid_t *ptid, struct user_desc *tls, pid_t *ctid */ );
```

Returns process ID of child on success, or –1 on error

Like *fork()*, a new process created with *clone()* is an almost exact duplicate of the parent.

Unlike *fork()*, the cloned child doesn't continue from the point of the call, but instead commences by calling the function specified in the *func* argument; we'll refer to this as the *child function*. When called, the child function is passed the value specified in *func_arg*. Using appropriate casting, the child function can freely interpret this argument; for example, as an *int* or as a pointer to a structure. (Interpreting it as a pointer is possible because the cloned child either obtains a copy of or shares the calling process's memory.)

> Within the kernel, *fork()*, *vfork()*, and *clone()* are ultimately implemented by the same function (*do_fork()* in `kernel/fork.c`). At this level, cloning is much closer to forking: *sys_clone()* doesn't have the *func* and *func_arg* arguments, and after the call, *sys_clone()* returns in the child in the same manner as *fork()*. The main text describes the *clone()* wrapper function that *glibc* provides for *sys_clone()*. (This function is defined in architecture-specific *glibc* assembler sources, such as in `sysdeps/unix/sysv/linux/i386/clone.S`.) This wrapper function invokes *func* after *sys_clone()* returns in the child.

The cloned child process terminates either when *func* returns (in which case its return value is the exit status of the process) or when the process makes a call to *exit()* (or *_exit()*). The parent process can wait for the cloned child in the usual manner using *wait()* or similar.

Since a cloned child may (like *vfork()*) share the parent's memory, it can't use the parent's stack. Instead, the caller must allocate a suitably sized block of memory for use as the child's stack and pass a pointer to that block in the argument *child_stack*. On most hardware architectures, the stack grows downward, so the *child_stack* argument should point to the high end of the allocated block.

> The architecture-dependence on the direction of stack growth is a defect in the design of *clone()*. On the Intel IA-64 architecture, an improved clone API is provided, in the form of *clone2()*. This system call defines the range of the stack of the child in a way that doesn't depend on the direction of stack growth, by supplying both the start address and size of the stack. See the manual page for details.

The *clone() flags* argument serves two purposes. First, its lower byte specifies the child's *termination signal*, which is the signal to be sent to the parent when the child terminates. (If a cloned child is *stopped* by a signal, the parent still receives `SIGCHLD`.) This byte may be 0, in which case no signal is generated. (Using the Linux-specific `/proc/`*PID*`/stat` file, we can determine the termination signal of any process; see the *proc(5)* manual page for further details.)

With *fork()* and *vfork()*, we have no way to select the termination signal; it is always `SIGCHLD`.

The remaining bytes of the *flags* argument hold a bit mask that controls the operation of *clone()*. We summarize these bit-mask values in Table 28-2, and describe them in more detail in Section 28.2.1.

Table 28-2: The *clone() flags* bit-mask values

Flag	**Effect if present**
`CLONE_CHILD_CLEARTID`	Clear *ctid* when child calls *exec()* or *_exit()* (2.6 onward)
`CLONE_CHILD_SETTID`	Write thread ID of child into *ctid* (2.6 onward)
`CLONE_FILES`	Parent and child share table of open file descriptors
`CLONE_FS`	Parent and child share attributes related to file system
`CLONE_IO`	Child shares parent's I/O context (2.6.25 onward)
`CLONE_NEWIPC`	Child gets new System V IPC namespace (2.6.19 onward)
`CLONE_NEWNET`	Child gets new network namespace (2.4.24 onward)
`CLONE_NEWNS`	Child gets copy of parent's mount namespace (2.4.19 onward)
`CLONE_NEWPID`	Child gets new process-ID namespace (2.6.19 onward)
`CLONE_NEWUSER`	Child gets new user-ID namespace (2.6.23 onward)
`CLONE_NEWUTS`	Child gets new UTS (*utsname()*) namespace (2.6.19 onward)
`CLONE_PARENT`	Make child's parent same as caller's parent (2.4 onward)
`CLONE_PARENT_SETTID`	Write thread ID of child into *ptid* (2.6 onward)
`CLONE_PID`	Obsolete flag used only by system boot process (up to 2.4)
`CLONE_PTRACE`	If parent is being traced, then trace child also
`CLONE_SETTLS`	*tls* describes thread-local storage for child (2.6 onward)
`CLONE_SIGHAND`	Parent and child share signal dispositions
`CLONE_SYSVSEM`	Parent and child share semaphore undo values (2.6 onward)
`CLONE_THREAD`	Place child in same thread group as parent (2.4 onward)
`CLONE_UNTRACED`	Can't force `CLONE_PTRACE` on child (2.6 onward)
`CLONE_VFORK`	Parent is suspended until child calls *exec()* or *_exit()*
`CLONE_VM`	Parent and child share virtual memory

The remaining arguments to *clone()* are *ptid*, *tls*, and *ctid*. These arguments relate to the implementation of threads, in particular the use of thread IDs and thread-local storage. We cover the use of these arguments when describing the *flags* bit-mask values in Section 28.2.1. (In Linux 2.4 and earlier, these three arguments are not provided by *clone()*. They were specifically added in Linux 2.6 to support the NPTL POSIX threads implementation.)

Example program

Listing 28-3 shows a simple example of the use of *clone()* to create a child process. The main program does the following:

- Open a file descriptor (for `/dev/null`) that will be closed by the child ②.
- Set the value for the *clone() flags* argument to `CLONE_FILES` ③ if a command-line argument was supplied, so that the parent and child will share a single file descriptor table. If no command-line argument was supplied, *flags* is set to 0.

- Allocate a stack for use by the child ④.
- If `CHILD_SIG` is nonzero and is not equal to `SIGCHLD`, ignore it, in case it is a signal that would terminate the process. We don't ignore `SIGCHLD`, because doing so would prevent waiting on the child to collect its status.
- Call *clone()* to create the child ⑥. The third (bit-mask) argument includes the termination signal. The fourth argument (*func_arg*) specifies the file descriptor opened earlier (at ②).
- Wait for the child to terminate ⑦.
- Check whether the file descriptor (opened at ②) is still open by trying to *write()* to it ⑧. The program reports whether the *write()* succeeds or fails.

Execution of the cloned child begins in *childFunc()*, which receives (in the argument *arg*) the file descriptor opened by the main program (at ②). The child closes this file descriptor and then terminates by performing a `return` ①.

Listing 28-3: Using *clone()* to create a child process

procexec/t_clone.c

```
#define _GNU_SOURCE
#include <signal.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <sched.h>
#include "tlpi_hdr.h"

#ifndef CHILD_SIG
#define CHILD_SIG SIGUSR1       /* Signal to be generated on termination
                                   of cloned child */
#endif

static int                      /* Startup function for cloned child */
childFunc(void *arg)
{
①   if (close(*((int *) arg)) == -1)
        errExit("close");

    return 0;                           /* Child terminates now */
}

int
main(int argc, char *argv[])
{
    const int STACK_SIZE = 65536;       /* Stack size for cloned child */
    char *stack;                        /* Start of stack buffer */
    char *stackTop;                     /* End of stack buffer */
    int s, fd, flags;

②   fd = open("/dev/null", O_RDWR);     /* Child will close this fd */
    if (fd == -1)
        errExit("open");
    /* If argc > 1, child shares file descriptor table with parent */

③   flags = (argc > 1) ? CLONE_FILES : 0;

    /* Allocate stack for child */

④   stack = malloc(STACK_SIZE);
    if (stack == NULL)
        errExit("malloc");
    stackTop = stack + STACK_SIZE;      /* Assume stack grows downward */

    /* Ignore CHILD_SIG, in case it is a signal whose default is to
       terminate the process; but don't ignore SIGCHLD (which is ignored
       by default), since that would prevent the creation of a zombie. */

⑤   if (CHILD_SIG != 0 && CHILD_SIG != SIGCHLD)
        if (signal(CHILD_SIG, SIG_IGN) == SIG_ERR)
            errExit("signal");

    /* Create child; child commences execution in childFunc() */

⑥   if (clone(childFunc, stackTop, flags | CHILD_SIG, (void *) &fd) == -1)
        errExit("clone");

    /* Parent falls through to here. Wait for child; __WCLONE is
       needed for child notifying with signal other than SIGCHLD. */

⑦   if (waitpid(-1, NULL, (CHILD_SIG != SIGCHLD) ? __WCLONE : 0) == -1)
        errExit("waitpid");
    printf("child has terminated\n");

    /* Did close() of file descriptor in child affect parent? */

⑧   s = write(fd, "x", 1);
    if (s == -1 && errno == EBADF)
        printf("file descriptor %d has been closed\n", fd);
    else if (s == -1)
        printf("write() on file descriptor %d failed "
                "unexpectedly (%s)\n", fd, strerror(errno));
    else
        printf("write() on file descriptor %d succeeded\n", fd);

    exit(EXIT_SUCCESS);
}
```

procexec/t_clone.c

When we run the program in Listing 28-3 without a command-line argument, we see the following:

```
$ ./t_clone                              Doesn't use CLONE_FILES
child has terminated
write() on file descriptor 3 succeeded   Child's close() did not affect parent
```

When we run the program with a command-line argument, we can see that the two processes share the file descriptor table:

```
$ ./t_clone x                            Uses CLONE_FILES
child has terminated
file descriptor 3 has been closed        Child's close() affected parent
```

> We show a more complex example of the use of *clone()* in the file `procexec/demo_clone.c` in the source code distribution for this book.

28.2.1 The *clone() flags* Argument

The *clone() flags* argument is a combination (ORing) of the bit-mask values described in the following pages. Rather than presenting these flags in alphabetical order, we present them in an order that eases explanation, and begin with those flags that are used in the implementation of POSIX threads. From the point of view of implementing threads, many uses of the word *process* below can be replaced by *thread*.

At this point, it is worth remarking that, to some extent, we are playing with words when trying to draw a distinction between the terms *thread* and *process*. It helps a little to introduce the term *kernel scheduling entity* (KSE), which is used in some texts to refer to the objects that are dealt with by the kernel scheduler. Really, threads and processes are simply KSEs that provide for greater and lesser degrees of sharing of attributes (virtual memory, open file descriptors, signal dispositions, process ID, and so on) with other KSEs. The POSIX threads specification provides just one out of various possible definitions of which attributes should be shared between threads.

In the course of the following descriptions, we'll sometimes mention the two main implementations of POSIX threads available on Linux: the older LinuxThreads implementation and the more recent NPTL implementation. Further information about these two implementations can be found in Section 33.5.

> Starting in kernel 2.6.16, Linux provides a new system call, *unshare()*, which allows a child created using *clone()* (or *fork()* or *vfork()*) to undo some of the attribute sharing (i.e., reverse the effects of some of the *clone() flags* bits) that was established when the child was created. For details, see the *unshare(2)* manual page.

Sharing file descriptor tables: `CLONE_FILES`

If the `CLONE_FILES` flag is specified, the parent and the child share the same table of open file descriptors. This means that file descriptor allocation or deallocation (*open()*, *close()*, *dup()*, *pipe()*, *socket()*, and so on) in either process will be visible in the other process. If the `CLONE_FILES` flag is not set, then the file descriptor table is not shared, and the child gets a copy of the parent's table at the time of the *clone()* call. These copied descriptors refer to the same open file descriptions as the corresponding descriptors in the parent (as with *fork()* and *vfork()*).

The specification of POSIX threads requires that all of the threads in a process share the same open file descriptors.

Sharing file system–related information: CLONE_FS

If the CLONE_FS flag is specified, then the parent and the child share file system–related information—umask, root directory, and current working directory. This means that calls to *umask()*, *chdir()*, or *chroot()* in either process will affect the other process. If the CLONE_FS flag is not set, then the parent and child have separate copies of this information (as with *fork()* and *vfork()*).

The attribute sharing provided by CLONE_FS is required by POSIX threads.

Sharing signal dispositions: CLONE_SIGHAND

If the CLONE_SIGHAND flag is set, then the parent and child share the same table of signal dispositions. Using *sigaction()* or *signal()* to change a signal's disposition in either process will affect that signal's disposition in the other process. If the CLONE_SIGHAND flag is not set, then signal dispositions are not shared; instead, the child gets a copy of the parent's signal disposition table (as with *fork()* and *vfork()*). The CLONE_SIGHAND flag doesn't affect the process signal mask and the set of pending signals, which are always distinct for the two processes. From Linux 2.6 onward, CLONE_VM must also be included in *flags* if CLONE_SIGHAND is specified.

Sharing of signal dispositions is required by POSIX threads.

Sharing the parent's virtual memory: CLONE_VM

If the CLONE_VM flag is set, then the parent and child share the same virtual memory pages (as with *vfork()*). Updates to memory or calls to *mmap()* or *munmap()* by either process will be visible to the other process. If the CLONE_VM flag is not set, then the child receives a copy of the parent's virtual memory (as with *fork()*).

Sharing the same virtual memory is one of the defining attributes of threads, and is required by POSIX threads.

Thread groups: CLONE_THREAD

If the CLONE_THREAD flag is set, then the child is placed in the same thread group as the parent. If this flag not set, the child is placed in its own new thread group.

Threads groups were introduced in Linux 2.4 to allow threading libraries to support the POSIX threads requirement that all of the threads in a process share a single process ID (i.e., *getpid()* in each of the threads should return the same value). A thread group is a group of KSEs that share the same *thread group identifier* (TGID), as shown in Figure 28-1. For the remainder of the discussion of CLONE_THREAD, we'll refer to these KSEs as *threads*.

Since Linux 2.4, *getpid()* returns the calling thread's TGID. In other words, a TGID is the same thing as a process ID.

> The *clone()* implementation in Linux 2.2 and earlier did not provide CLONE_THREAD. Instead, LinuxThreads implemented POSIX threads as processes that shared various attributes (e.g., virtual memory) but had distinct process IDs. For compatibility reasons, even on modern Linux kernels, the LinuxThreads implementation doesn't use the CLONE_THREAD flag, so that threads in that implementation continue to have distinct process IDs.

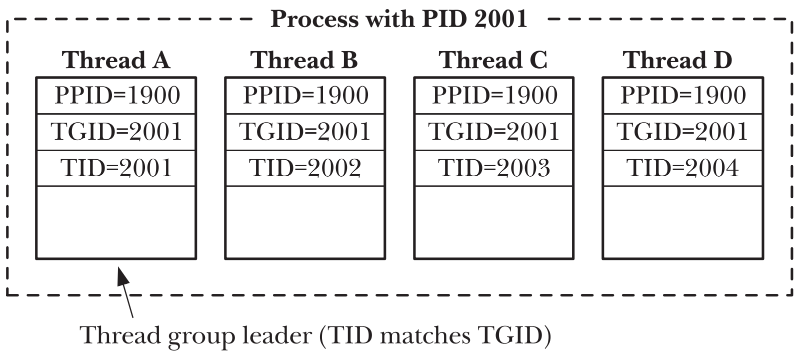

Figure 28-1: A thread group containing four threads

Each thread within a thread group is distinguished by a unique *thread identifier* (TID). Linux 2.4 introduced a new system call, *gettid()*, to allow a thread to obtain its own thread ID (this is the same value as is returned to the thread that calls *clone()*). A thread ID is represented using the same data type that is used for a process ID, *pid_t*. Thread IDs are unique system-wide, and the kernel guarantees that no thread ID will be the same as any process ID on the system, except when a thread is the thread group leader for a process.

The first thread in a new thread group has a thread ID that is the same as its thread group ID. This thread is referred to as the *thread group leader*.

> The thread IDs that we are discussing here are not the same as the thread IDs (the *pthread_t* data type) used by POSIX threads. The latter identifiers are generated and maintained internally (in user space) by a POSIX threads implementation.

All of the threads in a thread group have the same parent process ID—that of the thread group leader. Only after all of the threads in a thread group have terminated is a `SIGCHLD` signal (or other termination signal) sent to that parent process. These semantics correspond to the requirements of POSIX threads.

When a `CLONE_THREAD` thread terminates, no signal is sent to the thread that created it using *clone()*. Correspondingly, it is not possible to use *wait()* (or similar) to wait for a thread created using `CLONE_THREAD`. This accords with POSIX requirements. A POSIX thread is not the same thing as a process, and can't be waited for using *wait()*; instead, it must be joined using *pthread_join()*. To detect the termination of a thread created using `CLONE_THREAD`, a special synchronization primitive, called a *futex*, is used (see the discussion of the `CLONE_PARENT_SETTID` flag below).

If any of the threads in a thread group performs an *exec()*, then all threads other than the thread group leader are terminated (this behavior corresponds to the semantics required for POSIX threads), and the new program is execed in the thread group leader. In other words, in the new program, *gettid()* will return the thread ID of the thread group leader. During an *exec()*, the termination signal that this process should send to its parent is reset to `SIGCHLD`.

If one of the threads in a thread group creates a child using *fork()* or *vfork()*, then any thread in the group can monitor that child using *wait()* or similar.

From Linux 2.6 onward, `CLONE_SIGHAND` must also be included in *flags* if `CLONE_THREAD` is specified. This corresponds to further POSIX threads requirements;

for details, see the description of how POSIX threads and signals interact in Section 33.2. (The kernel handling of signals for CLONE_THREAD thread groups mirrors the POSIX requirements for how the threads in a process should react to signals.)

Threading library support: CLONE_PARENT_SETTID, CLONE_CHILD_SETTID, and CLONE_CHILD_CLEARTID

The CLONE_PARENT_SETTID, CLONE_CHILD_SETTID, and CLONE_CHILD_CLEARTID flags were added in Linux 2.6 to support the implementation of POSIX threads. These flags affect how *clone()* treats its *ptid* and *ctid* arguments. CLONE_PARENT_SETTID and CLONE_CHILD_CLEARTID are used in the NPTL threading implementation.

If the CLONE_PARENT_SETTID flag is set, then the kernel writes the thread ID of the child thread into the location pointed to by *ptid*. The thread ID is copied into *ptid* before the memory of the parent is duplicated. This means that, even if the CLONE_VM flag is not specified, both the parent and the child can see the child's thread ID in this location. (As noted above, the CLONE_VM flag is specified when creating POSIX threads.)

The CLONE_PARENT_SETTID flag exists in order to provide a reliable means for a threading implementation to obtain the ID of the new thread. Note that it isn't sufficient to obtain the thread ID of the new thread via the return value of *clone()*, like so:

```
tid = clone(...);
```

The problem is that this code can lead to various race conditions, because the assignment occurs only after *clone()* returns. For example, suppose that the new thread terminates, and the handler for its termination signal is invoked before the assignment to *tid* completes. In this case, the handler can't usefully access *tid*. (Within a threading library, *tid* might be an item in a global bookkeeping structure used to track the status of all threads.) Programs that invoke *clone()* directly often can be designed to work around this race condition. However, a threading library can't control the actions of the program that calls it. Using CLONE_PARENT_SETTID to ensure that the new thread ID is placed in the location pointed to by *ptid* before *clone()* returns allows a threading library to avoid such race conditions.

If the CLONE_CHILD_SETTID flag is set, then *clone()* writes the thread ID of the child thread into the location pointed to by *ctid*. The setting of *ctid* is done only in the child's memory, but this will affect the parent if CLONE_VM is also specified. Although NPTL doesn't need CLONE_CHILD_SETTID, this flag is provided to allow flexibility for other possible threading library implementations.

If the CLONE_CHILD_CLEARTID flag is set, then *clone()* zeros the memory location pointed to by *ctid* when the child terminates.

The *ctid* argument is the mechanism (described in a moment) by which the NPTL threading implementation obtains notification of the termination of a thread. Such notification is required by the *pthread_join()* function, which is the POSIX threads mechanism by which one thread can wait for the termination of another thread.

When a thread is created using *pthread_create()*, NPTL makes a *clone()* call in which *ptid* and *ctid* point to the same location. (This is why CLONE_CHILD_SETTID is not required by NPTL.) The CLONE_PARENT_SETTID flag causes that location to be initialized with the new thread's ID. When the child terminates and *ctid* is cleared, that change is visible to all threads in the process (since the `CLONE_VM` flag is also specified).

The kernel treats the location pointed to by *ctid* as a *futex*, an efficient synchronization mechanism. (See the *futex(2)* manual page for further details of futexes.) Notification of thread termination can be obtained by performing a *futex()* system call that blocks waiting for a change in the value at the location pointed to by *ctid*. (Behind the scenes, this is what *pthread_join()* does.) At the same time that the kernel clears *ctid*, it also wakes up any kernel scheduling entity (i.e., thread) that is blocked performing a futex wait on that address. (At the POSIX threads level, this causes the *pthread_join()* call to unblock.)

Thread-local storage: `CLONE_SETTLS`

If the `CLONE_SETTLS` flag is set, then the *tls* argument points to a *user_desc* structure describing the thread-local storage buffer to be used for this thread. This flag was added in Linux 2.6 to support the NPTL implementation of thread-local storage (Section 31.4). For details of the *user_desc* structure, see the definition and use of this structure in the 2.6 kernel sources and the *set_thread_area(2)* manual page.

Sharing System V semaphore undo values: `CLONE_SYSVSEM`

If the `CLONE_SYSVSEM` flag is set, then the parent and child share a single list of System V semaphore undo values (Section 47.8). If this flag is not set, then the parent and child have separate undo lists, and the child's undo list is initially empty.

The `CLONE_SYSVSEM` flag is available from kernel 2.6 onward, and provides the sharing semantics required by POSIX threads.

Per-process mount namespaces: `CLONE_NEWNS`

From kernel 2.4.19 onward, Linux supports the notion of per-process *mount namespaces*. A mount namespace is the set of mount points maintained by calls to *mount()* and *umount()*. The mount namespace affects how pathnames are resolved to actual files, as well as the operation of system calls such as *chdir()* and *chroot()*.

By default, the parent and the child share a mount namespace, which means that changes to the namespace by one process using *mount()* and *umount()* are visible in the other process (as with *fork()* and *vfork()*). A privileged (`CAP_SYS_ADMIN`) process may specify the `CLONE_NEWNS` flag so that the child obtains a copy of the parent's mount namespace. Thereafter, changes to the namespace by one process are not visible in the other process. (In earlier 2.4.*x* kernels, as well as in older kernels, we can consider all processes on the system as sharing a single system-wide mount namespace.)

Per-process mount namespaces can be used to create environments that are similar to *chroot()* jails, but which are more secure and flexible; for example, a jailed process can be provided with a mount point that is not visible to other processes on the system. Mount namespaces are also useful in setting up virtual server environments.

Specifying both `CLONE_NEWNS` and `CLONE_FS` in the same call to *clone()* is nonsensical and is not permitted.

Making the child's parent the same as the caller's: `CLONE_PARENT`

By default, when we create a new process with *clone()*, the parent of that process (as returned by *getppid()*) is the process that calls *clone()* (as with *fork()* and *vfork()*). If the `CLONE_PARENT` flag is set, then the parent of the child will be the caller's parent. In other words, `CLONE_PARENT` is the equivalent of setting *child.PPID = caller.PPID*. (In the default case, without `CLONE_PARENT`, it would be *child.PPID = caller.PID*.) The parent process (*child.PPID*) is the process that is signaled when the child terminates.

The `CLONE_PARENT` flag is available in Linux 2.4 and later. Originally, it was designed to be useful for POSIX threads implementations, but the 2.6 kernel pursued an approach to supporting threads (the use of `CLONE_THREAD`, described above) that removed the need for this flag.

Making the child's PID the same as the parent's PID: `CLONE_PID` (obsolete)

If the `CLONE_PID` flag is set, then the child has the same process ID as the parent. If this flag is not set, then the parent and child have different process IDs (as with *fork()* and *vfork()*). Only the system boot process (process ID 0) may specify this flag; it is used when initializing a multiprocessor system.

The `CLONE_PID` flag is not intended for use in user applications. In Linux 2.6, it has been removed, and is superseded by `CLONE_IDLETASK`, which causes the process ID of the new process to be set to 0. `CLONE_IDLETASK` is available only for internal use within the kernel (if specified in the *flags* argument of *clone()*, it is ignored). It is used to create the invisible per-CPU *idle process*, of which multiple instances may exist on multiprocessor systems.

Process tracing: `CLONE_PTRACE` and `CLONE_UNTRACED`

If the `CLONE_PTRACE` flag is set and the calling process is being traced, then the child is also traced. For details on process tracing (used by debuggers and the *strace* command), refer to the *ptrace(2)* manual page.

From kernel 2.6 onward, the `CLONE_UNTRACED` flag can be set, meaning that a tracing process can't force `CLONE_PTRACE` on this child. The `CLONE_UNTRACED` flag is used internally by the kernel in the creation of kernel threads.

Suspending the parent until the child exits or execs: `CLONE_VFORK`

If the `CLONE_VFORK` flag is set, then the execution of the parent is suspended until the child releases its virtual memory resources via a call to *exec()* or *_exit()* (as with *vfork()*).

New *clone()* flags to support containers

A number of new *clone() flags* values were added in Linux 2.6.19 and later: `CLONE_IO`, `CLONE_NEWIPC`, `CLONE_NEWNET`, `CLONE_NEWPID`, `CLONE_NEWUSER`, and `CLONE_NEWUTS`. (See the *clone(2)* manual page for the details of these flags.)

Most of these flags are provided to support the implementation of *containers* ([Bhattiprolu et al., 2008]). A container is a form of lightweight virtualization, whereby groups of processes running on the same kernel can be isolated from one another in environments that appear to be separate machines. Containers can also be nested, one inside the other. The containers approach contrasts with full virtualization, where each virtualized environment is running a distinct kernel.

To implement containers, the kernel developers had to provide a layer of indirection within the kernel around each of the global system resources—such as process IDs, the networking stack, the identifiers returned by *uname()*, System V IPC objects, and user and group ID namespaces—so that each container can provide its own instance of these resources.

There are various possible uses for containers, including the following:

- controlling allocation of resources on the system, such as network bandwidth or CPU time (e.g., one container might be granted 75% of the CPU time, while the other gets 25%);
- providing multiple lightweight virtual servers on a single host machine;
- freezing a container, so that execution of all processes in the container is suspended, later to be restarted, possibly after migrating to a different machine; and
- allowing an application's state to be dumped (checkpointed) and then later restored (perhaps after an application crash, or a planned or unplanned system shutdown) to continue computation from the time of the checkpoint.

Use of *clone() flags*

Roughly, we can say that a *fork()* corresponds to a *clone()* call with *flags* specified as just SIGCHLD, while a *vfork()* corresponds to a *clone()* call specifying *flags* as follows:

```
CLONE_VM | CLONE_VFORK | SIGCHLD
```

> Since version 2.3.3, the *glibc* wrapper *fork()* provided as part of the NPTL threading implementation bypasses the kernel's *fork()* system call and invokes *clone()*. This wrapper function invokes any fork handlers that have been established by the caller using *pthread_atfork()* (see Section 33.3).

The LinuxThreads threading implementation uses *clone()* (with just the first four arguments) to create threads by specifying *flags* as follows:

```
CLONE_VM | CLONE_FILES | CLONE_FS | CLONE_SIGHAND
```

The NPTL threading implementation uses *clone()* (with all seven arguments) to create threads by specifying *flags* as follows:

```
CLONE_VM | CLONE_FILES | CLONE_FS | CLONE_SIGHAND | CLONE_THREAD |
CLONE_SETTLS | CLONE_PARENT_SETTID | CLONE_CHILD_CLEARTID | CLONE_SYSVSEM
```

28.2.2 Extensions to *waitpid()* for Cloned Children

To wait for children produced by *clone()*, the following additional (Linux-specific) values can be included in the *options* bit-mask argument for *waitpid()*, *wait3()*, and *wait4()*:

__WCLONE
: If set, then wait for *clone* children only. If not set, then wait for *nonclone* children only. In this context, a *clone* child is one that delivers a signal other than SIGCHLD to its parent on termination. This bit is ignored if __WALL is also specified.

`__WALL` (since Linux 2.4)

Wait for all children, regardless of type (*clone* or *nonclone*).

`__WNOTHREAD` (since Linux 2.4)

By default, the wait calls wait not only for children of the calling process, but also for children of any other processes in the same thread group as the caller. Specifying the `__WNOTHREAD` flag limits the wait to children of the calling process.

These flags can't be used with *waitid()*.

28.3 Speed of Process Creation

Table 28-3 shows some speed comparisons for different methods of process creation. The results were obtained using a test program that executed a loop that repeatedly created a child process and then waited for it to terminate. The table compares the various methods using three different process memory sizes, as indicated by the *Total virtual memory* value. The differences in memory size were simulated by having the program *malloc()* additional memory on the heap prior to performing the timings.

Values for process size (*Total virtual memory*) in Table 28-3 are taken from the *VSZ* value displayed by the command *ps –o "pid vsz cmd"*.

Table 28-3: Time required to create 100,000 processes using *fork()*, *vfork()*, and *clone()*

Method of process creation	Total Virtual Memory					
	1.70 MB		2.70 MB		11.70 MB	
	Time (secs)	Rate	Time (secs)	Rate	Time (secs)	Rate
fork()	22.27 (7.99)	4544	26.38 (8.98)	4135	126.93 (52.55)	1276
vfork()	3.52 (2.49)	28955	3.55 (2.50)	28621	3.53 (2.51)	28810
clone()	2.97 (2.14)	34333	2.98 (2.13)	34217	2.93 (2.10)	34688
fork() + *exec()*	135.72 (12.39)	764	146.15 (16.69)	719	260.34 (61.86)	435
vfork() + *exec()*	107.36 (6.27)	969	107.81 (6.35)	964	107.97 (6.38)	960

For each process size, two types of statistics are provided in Table 28-3:

- The first statistic consists of two time measurements. The main (larger) measurement is the total elapsed (real) time to perform 100,000 process creation operations. The second time, shown in parentheses, is the CPU time consumed by the parent process. Since these tests were run on an otherwise unloaded machine, the difference between the two time values represents the total time consumed by child processes created during the test.

- The second statistic for each test shows the rate at which processes were created per (real) second. Statistics shown are the average of 20 runs for each case, and were obtained using kernel 2.6.27 running on an x86-32 system.

The first three data rows show times for simple process creation (without execing a new program in the child). In each case, the child processes exit immediately after they are created, and the parent waits for each child to terminate before creating the next.

The first row contains values for the *fork()* system call. From the data, we can see that as a process gets larger, *fork()* takes longer. These time differences show the additional time required to duplicate increasingly large page tables for the child and mark all data, heap, and stack segment page entries as read-only. (No *pages* are copied, since the child doesn't modify its data or stack segments.)

The second data row provides the same statistics for *vfork()*. We see that as the process size increases, the times remain the same—because no page tables or pages are copied during a *vfork()*, the virtual memory size of the calling process has no effect. The difference between the *fork()* and *vfork()* statistics represents the total time required for copying process page tables in each case.

> Small variations in the *vfork()* and *clone()* values in Table 28-3 are due to sampling errors and scheduling variations. Even when creating processes up to 300 MB in size, times for these two system calls remained constant.

The third data row shows statistics for process creation using *clone()* with the following flags:

```
CLONE_VM | CLONE_VFORK | CLONE_FS | CLONE_SIGHAND | CLONE_FILES
```

The first two of these flags emulate the behavior of *vfork()*. The remaining flags specify that the parent and child should share their file-system attributes (umask, root directory, and current working directory), table of signal dispositions, and table of open file descriptors. The difference between the *clone()* and *vfork()* data represents the small amount of additional work performed in *vfork()* to copy this information into the child process. The cost of copying file-system attributes and the table of signal dispositions is constant. However, the cost of copying the table of open file descriptors varies according to the number of descriptors. For example, opening 100 file descriptors in the parent process raised the *vfork()* real time (in the first column of the table) from 3.52 to 5.04 seconds, but left times for *clone()* unaffected.

> The timings for *clone()* are for the *glibc clone()* wrapper function, rather than direct calls to *sys_clone()*. Other tests (not summarized here) revealed negligible timing differences between using *sys_clone()* and calling *clone()* with a child function that immediately exited.

The differences between *fork()* and *vfork()* are quite marked. However, the following points should be kept in mind:

- The final data column, where *vfork()* is more than 30 times faster than *fork()*, corresponds to a large process. Typical processes would lie somewhere closer to the first two columns of the table.
- Because the times required for process creation are typically much smaller than those required for an *exec()*, the differences are much less marked if a

fork() or *vfork()* is followed by an *exec()*. This is illustrated by the final pair of data rows in Table 28-3, where each child performs an *exec()*, rather than immediately exiting. The program execed was the *true* command (`/bin/true`, chosen because it produces no output). In this case, we see that the relative differences between *fork()* and *vfork()* are much lower.

> In fact, the data shown in Table 28-3 doesn't reveal the full cost of an *exec()*, because the child execs the same program in each loop of the test. As a result, the cost of disk I/O to read the program into memory is essentially eliminated, because the program will be read into the kernel buffer cache on the first *exec()*, and then remain there. If each loop of the test execed a different program (e.g., a differently named copy of the same program), then we would observe a greater cost for an *exec()*.

28.4 Effect of *exec()* and *fork()* on Process Attributes

A process has numerous attributes, some of which we have already described in earlier chapters, and others that we explore in later chapters. Regarding these attributes, two questions arise:

- What happens to these attributes when a process performs an *exec()*?
- Which attributes are inherited by a child when a *fork()* is performed?

Table 28-4 summarizes the answers to these questions. The *exec()* column indicates which attributes are preserved during an *exec()*. The *fork()* column indicates which attributes are inherited (or in some cases, shared) by a child after *fork()*. Other than the attributes indicated as being Linux-specific, all listed attributes appear in standard UNIX implementations, and their handling during *exec()* and *fork()* conforms to the requirements of SUSv3.

Table 28-4: Effect of *exec()* and *fork()* on process attributes

Process attribute	***exec()***	***fork()***	**Interfaces affecting attribute; additional notes**
Process address space			
Text segment	No	Shared	Child process shares text segment with parent.
Stack segment	No	Yes	Function entry/exit; *alloca()*, *longjmp()*, *siglongjmp()*.
Data and heap segments	No	Yes	*brk()*, *sbrk()*.
Environment variables	See notes	Yes	*putenv()*, *setenv()*; direct modification of *environ*. Overwritten by *execle()* and *execve()* and preserved by remaining *exec()* calls.
Memory mappings	No	Yes; see notes	*mmap()*, *munmap()*. A mapping's `MAP_NORESERVE` flag is inherited across *fork()*. Mappings that have been marked with *madvise(MADV_DONTFORK)* are not inherited across *fork()*.
Memory locks	No	No	*mlock()*, *munlock()*.

Table 28-4: Effect of *exec()* and *fork()* on process attributes (continued)

Process attribute	***exec()***	***fork()***	**Interfaces affecting attribute; additional notes**
Process identifiers and credentials			
Process ID	Yes	No	
Parent process ID	Yes	No	
Process group ID	Yes	Yes	*setpgid()*.
Session ID	Yes	Yes	*setsid()*.
Real IDs	Yes	Yes	*setuid()*, *setgid()*, and related calls.
Effective and saved set IDs	See notes	Yes	*setuid()*, *setgid()*, and related calls. Chapter 9 explains how *exec()* affects these IDs.
Supplementary group IDs	Yes	Yes	*setgroups()*, *initgroups()*.
Files, file I/O, and directories			
Open file descriptors	See notes	Yes	*open()*, *close()*, *dup()*, *pipe()*, *socket()*, and so on. File descriptors are preserved across *exec()* unless marked close-on-exec. Descriptors in child and parent refer to same open file descriptions; see Section 5.4.
Close-on-exec flag	Yes (if off)	Yes	*fcntl(F_SETFD)*.
File offsets	Yes	Shared	*lseek()*, *read()*, *write()*, *readv()*, *writev()*. Child shares file offsets with parent.
Open file status flags	Yes	Shared	*open()*, *fcntl(F_SETFL)*. Child shares open file status flags with parent.
Asynchronous I/O operations	See notes	No	*aio_read()*, *aio_write()*, and related calls. Outstanding operations are canceled during an *exec()*.
Directory streams	No	Yes; see notes	*opendir()*, *readdir()*. SUSv3 states that child gets a copy of parent's directory streams, but these copies may or may not share the directory stream position. On Linux, the directory stream position is not shared.
File system			
Current working directory	Yes	Yes	*chdir()*.
Root directory	Yes	Yes	*chroot()*.
File mode creation mask	Yes	Yes	*umask()*.
Signals			
Signal dispositions	See notes	Yes	*signal()*, *sigaction()*. During an *exec()*, signals with dispositions set to default or ignore are unchanged; caught signals revert to their default dispositions. See Section 27.5.
Signal mask	Yes	Yes	Signal delivery, *sigprocmask()*, *sigaction()*.
Pending signal set	Yes	No	Signal delivery; *raise()*, *kill()*, *sigqueue()*.
Alternate signal stack	No	Yes	*sigaltstack()*.

(continued)

Table 28-4: Effect of *exec()* and *fork()* on process attributes (continued)

Process attribute	***exec()***	***fork()***	**Interfaces affecting attribute; additional notes**
Timers			
Interval timers	Yes	No	*setitimer()*.
Timers set by *alarm()*	Yes	No	*alarm()*.
POSIX timers	No	No	*timer_create()* and related calls.
POSIX threads			
Threads	No	See notes	During *fork()*, only calling thread is replicated in child.
Thread cancelability state and type	No	Yes	After an *exec()*, the cancelability type and state are reset to `PTHREAD_CANCEL_ENABLE` and `PTHREAD_CANCEL_DEFERRED`, respectively
Mutexes and condition variables	No	Yes	See Section 33.3 for details of the treatment of mutexes and other thread resources during *fork()*.
Priority and scheduling			
Nice value	Yes	Yes	*nice()*, *setpriority()*.
Scheduling policy and priority	Yes	Yes	*sched_setscheduler()*, *sched_setparam()*.
Resources and CPU time			
Resource limits	Yes	Yes	*setrlimit()*.
Process and child CPU times	Yes	No	As returned by *times()*.
Resource usages	Yes	No	As returned by *getrusage()*.
Interprocess communication			
System V shared memory segments	No	Yes	*shmat()*, *shmdt()*.
POSIX shared memory	No	Yes	*shm_open()* and related calls.
POSIX message queues	No	Yes	*mq_open()* and related calls. Descriptors in child and parent refer to same open message queue descriptions. A child doesn't inherit its parent's message notification registrations.
POSIX named semaphores	No	Shared	*sem_open()* and related calls. Child shares references to same semaphores as parent.
POSIX unnamed semaphores	No	See notes	*sem_init()* and related calls. If semaphores are in a shared memory region, then child shares semaphores with parent; otherwise, child has its own copy of the semaphores.
System V semaphore adjustments	Yes	No	*semop()*. See Section 47.8.
File locks	Yes	See notes	*flock()*. Child inherits a reference to the same lock as parent.
Record locks	See notes	No	*fcntl(F_SETLK)*. Locks are preserved across *exec()* unless a file descriptor referring to the file is marked close-on-exec; see Section 55.3.5.

Table 28-4: Effect of *exec()* and *fork()* on process attributes (continued)

Process attribute	***exec()***	***fork()***	**Interfaces affecting attribute; additional notes**
Miscellaneous			
Locale settings	No	Yes	*setlocale()*. As part of C run-time initialization, the equivalent of *setlocale(LC_ALL, "C")* is executed after a new program is execed.
Floating-point environment	No	Yes	When a new program is execed, the state of the floating-point environment is reset to the default; see *fenv(3)*.
Controlling terminal	Yes	Yes	
Exit handlers	No	Yes	*atexit()*, *on_exit()*.
Linux-specific			
File-system IDs	See notes	Yes	*setfsuid()*, *setfsgid()*. These IDs are also changed any time the corresponding effective IDs are changed.
timerfd timers	Yes	See notes	*timerfd_create()*; child inherits file descriptors referring to same timers as parent.
Capabilities	See notes	Yes	*capset()*. The handling of capabilities during an *exec()* is described in Section 39.5.
Capability bounding set	Yes	Yes	
Capabilities *securebits* flags	See notes	Yes	All *securebits* flags are preserved during an *exec()* except `SECBIT_KEEP_CAPS`, which is always cleared.
CPU affinity	Yes	Yes	*sched_setaffinity()*.
`SCHED_RESET_ON_FORK`	Yes	No	See Section 35.3.2.
Allowed CPUs	Yes	Yes	See *cpuset(7)*.
Allowed memory nodes	Yes	Yes	See *cpuset(7)*.
Memory policy	Yes	Yes	See *set_mempolicy(2)*.
File leases	Yes	See notes	*fcntl(F_SETLEASE)*. Child inherits a reference to the same lease as parent.
Directory change notifications	Yes	No	The *dnotify* API, available via *fcntl(F_NOTIFY)*.
prctl(PR_SET_DUMPABLE)	See notes	Yes	During an *exec()*, the `PR_SET_DUMPABLE` flag is set, unless execing a set-user-ID or set-group-ID program, in which case it is cleared.
prctl(PR_SET_PDEATHSIG)	Yes	No	
prctl(PR_SET_NAME)	No	Yes	
`oom_adj`	Yes	Yes	See Section 49.9.
`coredump_filter`	Yes	Yes	See Section 22.1.

28.5 Summary

When process accounting is enabled, the kernel writes an accounting record to a file for each process that terminates on the system. This record contains statistics on the resources used by the process.

Like *fork()*, the Linux-specific *clone()* system call creates a new process, but allows finer control over which attributes are shared between the parent and child. This system call is used primarily for implementing threading libraries.

We compared the speed of process creation using *fork()*, *vfork()*, and *clone()*. Although *vfork()* is faster than *fork()*, the time difference between these system calls is small by comparison with the time required for a child process to do a subsequent *exec()*.

When a child process is created via *fork()*, it inherits copies of (or in some cases shares) certain process attributes from its parent, while other process attributes are not inherited. For example, a child inherits copies of its parent's file descriptor table and signal dispositions, but doesn't inherit its parent's interval timers, record locks, or set of pending signals. Correspondingly, when a process performs an *exec()*, certain process attributes remain unchanged, while others are reset to defaults. For example, the process ID remains the same, file descriptors remain open (unless marked close-on-exec), interval timers are preserved, and pending signals remain pending, but handled signals are reset to their default disposition and shared memory segments are detached.

Further information

Refer to the sources of further information listed in Section 24.6. Chapter 17 of [Frisch, 2002] describes the administration of process accounting, as well as some of the variations across UNIX implementations. [Bovet & Cesati, 2005] describes the implementation of the *clone()* system call.

28.6 Exercise

28-1. Write a program to see how fast the *fork()* and *vfork()* system calls are on your system. Each child process should immediately exit, and the parent should *wait()* on each child before creating the next. Compare the relative differences for these two system calls with those of Table 28-3. The shell built-in command *time* can be used to measure the execution time of a program.