26

MONITORING CHILD PROCESSES

In many application designs, a parent process needs to know when one of its child processes changes state—when the child terminates or is stopped by a signal. This chapter describes two techniques used to monitor child processes: the *wait()* system call (and its variants) and the use of the `SIGCHLD` signal.

26.1 Waiting on a Child Process

In many applications where a parent creates child processes, it is useful for the parent to be able to monitor the children to find out when and how they terminate. This facility is provided by *wait()* and a number of related system calls.

26.1.1 The *wait()* System Call

The *wait()* system call waits for one of the children of the calling process to terminate and returns the termination status of that child in the buffer pointed to by *status*.

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

pid_t wait(int *status);
                                Returns process ID of terminated child, or –1 on error
```

The *wait()* system call does the following:

1. If no (previously unwaited-for) child of the calling process has yet terminated, the call blocks until one of the children terminates. If a child has already terminated by the time of the call, *wait()* returns immediately.
2. If *status* is not NULL, information about how the child terminated is returned in the integer to which *status* points. We describe the information returned in *status* in Section 26.1.3.
3. The kernel adds the process CPU times (Section 10.7) and resource usage statistics (Section 36.1) to running totals for all children of this parent process.
4. As its function result, *wait()* returns the process ID of the child that has terminated.

On error, *wait()* returns –1. One possible error is that the calling process has no (previously unwaited-for) children, which is indicated by the *errno* value ECHILD. This means that we can use the following loop to wait for all children of the calling process to terminate:

```
while ((childPid = wait(NULL)) != -1)
    continue;
if (errno != ECHILD)                /* An unexpected error... */
    errExit("wait");
```

Listing 26-1 demonstrates the use of *wait()*. This program creates multiple child processes, one per (integer) command-line argument. Each child sleeps for the number of seconds specified in the corresponding command-line argument and then exits. In the meantime, after all children have been created, the parent process repeatedly calls *wait()* to monitor the termination of its children. This loop continues until *wait()* returns –1. (This is not the only approach: we could alternatively exit the loop when the number of terminated children, *numDead*, matches the number of children created.) The following shell session log shows what happens when we use the program to create three children:

```
$ ./multi_wait 7 1 4
[13:41:00] child 1 started with PID 21835, sleeping 7 seconds
[13:41:00] child 2 started with PID 21836, sleeping 1 seconds
[13:41:00] child 3 started with PID 21837, sleeping 4 seconds
[13:41:01] wait() returned child PID 21836 (numDead=1)
[13:41:04] wait() returned child PID 21837 (numDead=2)
[13:41:07] wait() returned child PID 21835 (numDead=3)
No more children - bye!
```

> If there are multiple terminated children at a particular moment, SUSv3 leaves unspecified the order in which these children will be reaped by a sequence of *wait()* calls; that is, the order depends on the implementation. Even across versions of the Linux kernel, the behavior varies.

Listing 26-1: Creating and waiting for multiple children

procexec/multi_wait.c

```
#include <sys/wait.h>
#include <time.h>
#include "curr_time.h"              /* Declaration of currTime() */
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int numDead;       /* Number of children so far waited for */
    pid_t childPid;    /* PID of waited for child */
    int j;

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s sleep-time...\n", argv[0]);

    setbuf(stdout, NULL);           /* Disable buffering of stdout */

    for (j = 1; j < argc; j++) {    /* Create one child for each argument */
        switch (fork()) {
        case -1:
            errExit("fork");

        case 0:                     /* Child sleeps for a while then exits */
            printf("[%s] child %d started with PID %ld, sleeping %s "
                    "seconds\n", currTime("%T"), j, (long) getpid(), argv[j]);
            sleep(getInt(argv[j], GN_NONNEG, "sleep-time"));
            _exit(EXIT_SUCCESS);

        default:                    /* Parent just continues around loop */
            break;
        }
    }

    numDead = 0;
    for (;;) {                      /* Parent waits for each child to exit */
        childPid = wait(NULL);
        if (childPid == -1) {
            if (errno == ECHILD) {
                printf("No more children - bye!\n");
                exit(EXIT_SUCCESS);
            } else {                /* Some other (unexpected) error */
                errExit("wait");
            }
        }

        numDead++;
        printf("[%s] wait() returned child PID %ld (numDead=%d)\n",
                currTime("%T"), (long) childPid, numDead);
    }
}
```

procexec/multi_wait.c

26.1.2 The *waitpid()* System Call

The *wait()* system call has a number of limitations, which *waitpid()* was designed to address:

- If a parent process has created multiple children, it is not possible to *wait()* for the completion of a specific child; we can only wait for the next child that terminates.
- If no child has yet terminated, *wait()* always blocks. Sometimes, it would be preferable to perform a nonblocking wait so that if no child has yet terminated, we obtain an immediate indication of this fact.
- Using *wait()*, we can find out only about children that have terminated. It is not possible to be notified when a child is stopped by a signal (such as `SIGSTOP` or `SIGTTIN`) or when a stopped child is resumed by delivery of a `SIGCONT` signal.

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

pid_t waitpid(pid_t pid, int *status, int options);
                                Returns process ID of child, 0 (see text), or –1 on error
```

The return value and *status* arguments of *waitpid()* are the same as for *wait()*. (See Section 26.1.3 for an explanation of the value returned in *status*.) The *pid* argument enables the selection of the child to be waited for, as follows:

- If *pid* is greater than 0, wait for the child whose *process ID* equals *pid*.
- If *pid* equals 0, wait for any child in the *same process group as the caller* (parent). We describe process groups in Section 34.2.
- If *pid* is less than –1, wait for any child whose *process group* identifier equals the absolute value of *pid*.
- If *pid* equals –1, wait for *any* child. The call *wait(&status)* is equivalent to the call *waitpid(–1, &status, 0)*.

The *options* argument is a bit mask that can include (OR) zero or more of the following flags (all of which are specified in SUSv3):

`WUNTRACED`
: In addition to returning information about terminated children, also return information when a child is *stopped* by a signal.

`WCONTINUED` (since Linux 2.6.10)
: Also return status information about stopped children that have been resumed by delivery of a `SIGCONT` signal.

`WNOHANG`
: If no child specified by *pid* has yet changed state, then return immediately, instead of blocking (i.e., perform a "poll"). In this case, the return value of *waitpid()* is 0. If the calling process has no children that match the specification in *pid*, *waitpid()* fails with the error `ECHILD`.

We demonstrate the use of *waitpid()* in Listing 26-3.

In its rationale for *waitpid()*, SUSv3 notes that the name WUNTRACED is a historical artifact of this flag's origin in BSD, where a process could be stopped in one of two ways: as a consequence of being traced by the *ptrace()* system call, or by being stopped by a signal (i.e., not being traced). When a child is being traced by *ptrace()*, then delivery of *any* signal (other than SIGKILL) causes the child to be stopped, and a SIGCHLD signal is consequently sent to the parent. This behavior occurs even if the child is ignoring the signal. However, if the child is blocking the signal, then it is not stopped (unless the signal is SIGSTOP, which can't be blocked).

26.1.3 The Wait Status Value

The *status* value returned by *wait()* and *waitpid()* allows us to distinguish the following events for the child:

- The child terminated by calling *_exit()* (or *exit()*), specifying an integer *exit status*.
- The child was terminated by the delivery of an unhandled signal.
- The child was stopped by a signal, and *waitpid()* was called with the WUNTRACED flag.
- The child was resumed by a SIGCONT signal, and *waitpid()* was called with the WCONTINUED flag.

We use the term *wait status* to encompass all of the above cases. The designation *termination status* is used to refer to the first two cases. (In the shell, we can obtain the termination status of the last command executed by examining the contents of the variable *$?*.)

Although defined as an *int*, only the bottom 2 bytes of the value pointed to by *status* are actually used. The way in which these 2 bytes are filled depends on which of the above events occurred for the child, as depicted in Figure 26-1.

Figure 26-1 shows the layout of the wait status value for Linux/x86-32. The details vary across implementations. SUSv3 doesn't specify any particular layout for this information, or even require that it is contained in the bottom 2 bytes of the value pointed to by *status*. Portable applications should always use the macros described in this section to inspect this value, rather than directly inspecting its bit-mask components.

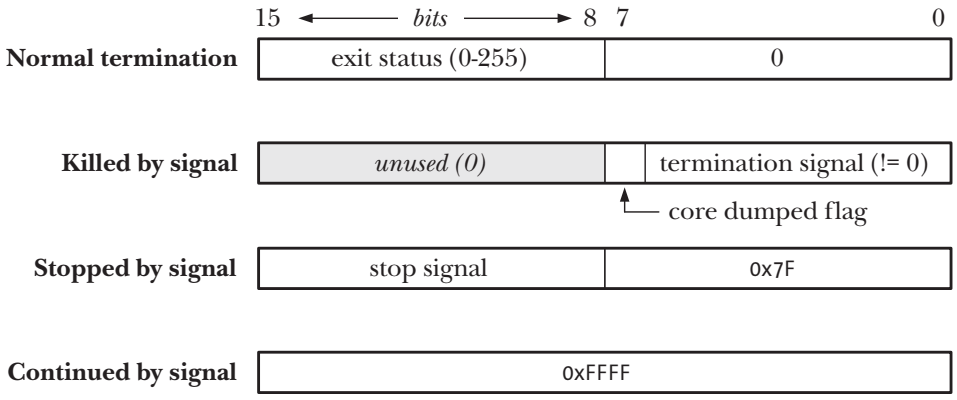

Figure 26-1: Value returned in the *status* argument of *wait()* and *waitpid()*

The <sys/wait.h> header file defines a standard set of macros that can be used to dissect a wait status value. When applied to a *status* value returned by *wait()* or *waitpid()*, only one of the macros in the list below will return true. Additional macros are provided to further dissect the *status* value, as noted in the list.

WIFEXITED(status)
: This macro returns true if the child process exited normally. In this case, the macro WEXITSTATUS(status) returns the exit status of the child process. (As noted in Section 25.1, only the least significant byte of the child's exit status is available to the parent.)

WIFSIGNALED(status)
: This macro returns true if the child process was killed by a signal. In this case, the macro WTERMSIG(status) returns the number of the signal that caused the process to terminate, and the macro WCOREDUMP(status) returns true if the child process produced a core dump file. The WCOREDUMP() macro is not specified by SUSv3, but is available on most UNIX implementations.

WIFSTOPPED(status)
: This macro returns true if the child process was stopped by a signal. In this case, the macro WSTOPSIG(status) returns the number of the signal that stopped the process.

WIFCONTINUED(status)
: This macro returns true if the child was resumed by delivery of SIGCONT. This macro is available since Linux 2.6.10.

Note that although the name *status* is also used for the argument of the above macros, they expect a plain integer, rather than a pointer to an integer as required by *wait()* and *waitpid()*.

Example program

The *printWaitStatus()* function of Listing 26-2 uses all of the macros described above. This function dissects and prints the contents of a wait status value.

Listing 26-2: Displaying the status value returned by *wait()* and related calls

procexec/print_wait_status.c

```
#define _GNU_SOURCE     /* Get strsignal() declaration from <string.h> */
#include <string.h>
#include <sys/wait.h>
#include "print_wait_status.h"  /* Declaration of printWaitStatus() */
#include "tlpi_hdr.h"

/* NOTE: The following function employs printf(), which is not
   async-signal-safe (see Section 21.1.2). As such, this function is
   also not async-signal-safe (i.e., beware of calling it from a
   SIGCHLD handler). */

void                    /* Examine a wait() status using the W* macros */
printWaitStatus(const char *msg, int status)
{
```

```
    if (msg != NULL)
        printf("%s", msg);

    if (WIFEXITED(status)) {
        printf("child exited, status=%d\n", WEXITSTATUS(status));

    } else if (WIFSIGNALED(status)) {
        printf("child killed by signal %d (%s)",
                WTERMSIG(status), strsignal(WTERMSIG(status)));
#ifdef WCOREDUMP        /* Not in SUSv3, may be absent on some systems */
        if (WCOREDUMP(status))
            printf(" (core dumped)");
#endif
        printf("\n");

    } else if (WIFSTOPPED(status)) {
        printf("child stopped by signal %d (%s)\n",
                WSTOPSIG(status), strsignal(WSTOPSIG(status)));

#ifdef WIFCONTINUED     /* SUSv3 has this, but older Linux versions and
                           some other UNIX implementations don't */
    } else if (WIFCONTINUED(status)) {
        printf("child continued\n");
#endif

    } else {            /* Should never happen */
        printf("what happened to this child? (status=%x)\n",
                (unsigned int) status);
    }
}
```

procexec/print_wait_status.c

The *printWaitStatus()* function is used in Listing 26-3. This program creates a child process that either loops continuously calling *pause()* (during which time signals can be sent to the child) or, if an integer command-line argument was supplied, exits immediately using this integer as the exit status. In the meantime, the parent monitors the child via *waitpid()*, printing the returned status value and passing this value to *printWaitStatus()*. The parent exits when it detects that the child has either exited normally or been terminated by a signal.

The following shell session shows a few example runs of the program in Listing 26-3. We begin by creating a child that immediately exits with a status of 23:

```
$ ./child_status 23
Child started with PID = 15807
waitpid() returned: PID=15807; status=0x1700 (23,0)
child exited, status=23
```

In the next run, we start the program in the background, and then send SIGSTOP and SIGCONT signals to the child:

```
$ ./child_status &
[1] 15870
$ Child started with PID = 15871
```

```
kill -STOP 15871
$ waitpid() returned: PID=15871; status=0x137f (19,127)
child stopped by signal 19 (Stopped (signal))
kill -CONT 15871
$ waitpid() returned: PID=15871; status=0xffff (255,255)
child continued
```

The last two lines of output will appear only on Linux 2.6.10 and later, since earlier kernels don't support the *waitpid()* WCONTINUED option. (This shell session is made slightly hard to read by the fact that output from the program executing in the background is in some cases intermingled with the prompt produced by the shell.)

We continue the shell session by sending a SIGABRT signal to terminate the child:

```
kill -ABRT 15871
$ waitpid() returned: PID=15871; status=0x0006 (0,6)
child killed by signal 6 (Aborted)
Press Enter, in order to see shell notification that background job has terminated
[1]+  Done              ./child_status
$ ls -l core
ls: core: No such file or directory
$ ulimit -c                                          Display RLIMIT_CORE limit
0
```

Although the default action of SIGABRT is to produce a core dump file and terminate the process, no core file was produced. This is because core dumps were disabled—the RLIMIT_CORE soft resource limit (Section 36.3), which specifies the maximum size of a core file, was set to 0, as shown by the *ulimit* command above.

We repeat the same experiment, but this time enabling core dumps before sending SIGABRT to the child:

```
$ ulimit -c unlimited                            Allow core dumps
$ ./child_status &
[1] 15902
$ Child started with PID = 15903
kill -ABRT 15903                                 Send SIGABRT to child
$ waitpid() returned: PID=15903; status=0x0086 (0,134)
child killed by signal 6 (Aborted) (core dumped)
Press Enter, in order to see shell notification that background job has terminated
[1]+  Done              ./child_status
$ ls -l core                                     This time we get a core dump
-rw-------   1 mtk      users       65536 May  6 21:01 core
```

Listing 26-3: Using *waitpid()* to retrieve the status of a child process

procexec/child_status.c

```
#include <sys/wait.h>
#include "print_wait_status.h"          /* Declares printWaitStatus() */
#include "tlpi_hdr.h"

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

```
    if (argc > 1 && strcmp(argv[1], "--help") == 0)
        usageErr("%s [exit-status]\n", argv[0]);

    switch (fork()) {
    case -1: errExit("fork");

    case 0:             /* Child: either exits immediately with given
                           status or loops waiting for signals */
        printf("Child started with PID = %ld\n", (long) getpid());
        if (argc > 1)                   /* Status supplied on command line? */
            exit(getInt(argv[1], 0, "exit-status"));
        else                            /* Otherwise, wait for signals */
            for (;;)
                pause();
        exit(EXIT_FAILURE);             /* Not reached, but good practice */

    default:            /* Parent: repeatedly wait on child until it
                           either exits or is terminated by a signal */
        for (;;) {
            childPid = waitpid(-1, &status, WUNTRACED
#ifdef WCONTINUED       /* Not present on older versions of Linux */
                                            | WCONTINUED
#endif
                       );
            if (childPid == -1)
                errExit("waitpid");

            /* Print status in hex, and as separate decimal bytes */

            printf("waitpid() returned: PID=%ld; status=0x%04x (%d,%d)\n",
                    (long) childPid,
                    (unsigned int) status, status >> 8, status & 0xff);
            printWaitStatus(NULL, status);

            if (WIFEXITED(status) || WIFSIGNALED(status))
                exit(EXIT_SUCCESS);
        }
    }
}
```

procexec/child_status.c

26.1.4 Process Termination from a Signal Handler

As shown in Table 20-1 (on page 396), some signals terminate a process by default. In some circumstances, we may wish to have certain cleanup steps performed before a process terminates. For this purpose, we can arrange to have a handler catch such signals, perform the cleanup steps, and then terminate the process. If we do this, we should bear in mind that the termination status of a process is available to its parent via *wait()* or *waitpid()*. For example, calling *_exit(EXIT_SUCCESS)* from the signal handler will make it appear to the parent process that the child terminated successfully.

If the child needs to inform the parent that it terminated because of a signal, then the child's signal handler should first disestablish itself, and then raise the

same signal once more, which this time will terminate the process. The signal handler would contain code such as the following:

```
void
handler(int sig)
{
    /* Perform cleanup steps */

    signal(sig, SIG_DFL);           /* Disestablish handler */
    raise(sig);                     /* Raise signal again */
}
```

26.1.5 The *waitid()* System Call

Like *waitpid()*, *waitid()* returns the status of child processes. However, *waitid()* provides extra functionality that is unavailable with *waitpid()*. This system call derives from System V, but is now specified in SUSv3. It was added to Linux in kernel 2.6.9.

> Before Linux 2.6.9, a version of *waitid()* was provided via an implementation in *glibc*. However, because a full implementation of this interface requires kernel support, the *glibc* implementation provided no more functionality than was available using *waitpid()*.

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

int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

Returns 0 on success or if WNOHANG was specified and there were no children to wait for, or −1 on error

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

- If *idtype* is P_ALL, wait for any child; *id* is ignored.
- If *idtype* is P_PID, wait for the child whose process ID equals *id*.
- If *idtype* is P_PGID, wait for any child whose process group ID equals *id*.

> Note that unlike *waitpid()*, it is not possible to specify 0 in *id* to mean any process in the same process group as the caller. Instead, we must explicitly specify the caller's process group ID using the value returned by *getpgrp()*.

The most significant difference between *waitpid()* and *waitid()* is that *waitid()* provides more precise control of the child events that should be waited for. We control this by ORing one or more of the following flags in *options*:

WEXITED
: Wait for children that have terminated, either normally or abnormally.

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

WCONTINUED
: Wait for children that have been resumed by a SIGCONT signal.

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

WNOHANG
: This flag has the same meaning as for *waitpid()*. If none of the children matching the specification in *id* has status information to return, then return immediately (a poll). In this case, the return value of *waitid()* is 0. If the calling process has no children that match the specification in *id*, *waitid()* instead fails with the error ECHILD.

WNOWAIT
: Normally, once a child has been waited for using *waitid()*, then that "status event" is consumed. However, if WNOWAIT is specified, then the child status is returned, but the child remains in a waitable state, and we can later wait for it again to retrieve the same information.

On success, *waitid()* returns 0, and the *siginfo_t* structure (Section 21.4) pointed to by *infop* is updated to contain information about the child. The following fields are filled in the *siginfo_t* structure:

si_code
: This field contains one of the following values: CLD_EXITED, indicating that the child terminated by calling *_exit()*; CLD_KILLED, indicating that the child was killed by a signal; CLD_STOPPED, indicating that the child was stopped by a signal; or CLD_CONTINUED, indicating that the (previously stopped) child resumed execution as a consequence of receiving a (SIGCONT) signal.

si_pid
: This field contains the process ID of the child whose state has changed.

si_signo
: This field is always set to SIGCHLD.

si_status
: This field contains either the exit status of the child, as passed to *_exit()*, or the signal that caused the child to stop, continue, or terminate. We can determine which type of information is in this field by examining the *si_code* field.

si_uid
: This field contains the real user ID of the child. Most other UNIX implementations don't set this field.

> On Solaris, two additional fields are filled in: *si_stime* and *si_utime*. These contain the system and user CPU time used by the child, respectively. SUSv3 doesn't require these fields to be set by *waitid()*.

One detail of the operation of *waitid()* needs further clarification. If WNOHANG is specified in *options*, then a 0 return value from *waitid()* can mean one of two things: a child had already changed state at the time of the call (and information about the child is returned in the *siginfo_t* structure pointed to by *infop*), or there was no child whose state has changed. For the case where no child has changed state, some UNIX implementations (including Linux), zero out the returned *siginfo_t* structure. This provides a method of distinguishing the two possibilities: we can check whether the

value in *si_pid* is 0 or nonzero. Unfortunately, this behavior is not required by SUSv3, and some UNIX implementations leave the *siginfo_t* structure unchanged in this case. (A future corrigendum to SUSv4 is likely to add a requirement that *si_pid* and *si_signo* are zeroed in this case.) The only portable way to distinguish these two cases is to zero out the *siginfo_t* structure before calling *waitid()*, as in the following code:

```
siginfo_t info;
...
memset(&info, 0, sizeof(siginfo_t));
if (waitid(idtype, id, &info, options | WNOHANG) == -1)
    errExit("waitid");
if (info.si_pid == 0) {
    /* No children changed state */
} else {
    /* A child changed state; details are provided in 'info' */
}
```

26.1.6 The *wait3()* and *wait4()* System Calls

The *wait3()* and *wait4()* system calls perform a similar task to *waitpid()*. The principal semantic difference is that *wait3()* and *wait4()* return *resource usage* information about the terminated child in the structure pointed to by the *rusage* argument. This information includes the amount of CPU time used by the process and memory-management statistics. We defer detailed discussion of the *rusage* structure until Section 36.1, where we describe the *getrusage()* system call.

```
#define _BSD_SOURCE       /* Or #define _XOPEN_SOURCE 500 for wait3() */
#include <sys/resource.h>
#include <sys/wait.h>

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

Both return process ID of child, or −1 on error

Excluding the use of the *rusage* argument, a call to *wait3()* is equivalent to the following *waitpid()* call:

```
waitpid(-1, &status, options);
```

Similarly, *wait4()* is equivalent to the following:

```
waitpid(pid, &status, options);
```

In other words, *wait3()* waits for any child, while *wait4()* can be used to select a specific child or children upon which to wait.

On some UNIX implementations, *wait3()* and *wait4()* return resource usage information only for terminated children. On Linux, resource usage information can also be retrieved for stopped children if the WUNTRACED flag is specified in *options*.

The names for these two system calls refer to the number of arguments they each take. Both system calls originated in BSD, but are now available on most

UNIX implementations. Neither is standardized in SUSv3. (SUSv2 did specify *wait3()*, but marked it LEGACY.)

We usually avoid the use of *wait3()* and *wait4()* in this book. Typically, we don't need the extra information returned by these calls. Also, lack of standardization limits their portability.

26.2 Orphans and Zombies

The lifetimes of parent and child processes are usually not the same—either the parent outlives the child or vice versa. This raises two questions:

- Who becomes the parent of an *orphaned* child? The orphaned child is adopted by *init*, the ancestor of all processes, whose process ID is 1. In other words, after a child's parent terminates, a call to *getppid()* will return the value 1. This can be used as a way of determining if a child's true parent is still alive (this assumes a child that was created by a process other than *init*).

  > Using the `PR_SET_PDEATHSIG` operation of the Linux-specific *prctl()* system call, it is possible to arrange that a process receives a specified signal when it becomes an orphan.

- What happens to a child that terminates before its parent has had a chance to perform a *wait()*? The point here is that, although the child has finished its work, the parent should still be permitted to perform a *wait()* at some later time to determine how the child terminated. The kernel deals with this situation by turning the child into a *zombie*. This means that most of the resources held by the child are released back to the system to be reused by other processes. The only part of the process that remains is an entry in the kernel's process table recording (among other things) the child's process ID, termination status, and resource usage statistics (Section 36.1).

Regarding zombies, UNIX systems imitate the movies—a zombie process can't be killed by a signal, not even the (silver bullet) `SIGKILL`. This ensures that the parent can always eventually perform a *wait()*.

When the parent does perform a *wait()*, the kernel removes the zombie, since the last remaining information about the child is no longer required. On the other hand, if the parent terminates without doing a *wait()*, then the *init* process adopts the child and automatically performs a *wait()*, thus removing the zombie process from the system.

If a parent creates a child, but fails to perform a *wait()*, then an entry for the zombie child will be maintained indefinitely in the kernel's process table. If a large number of such zombie children are created, they will eventually fill the kernel process table, preventing the creation of new processes. Since the zombies can't be killed by a signal, the only way to remove them from the system is to kill their parent (or wait for it to exit), at which time the zombies are adopted and waited on by *init*, and consequently removed from the system.

These semantics have important implications for the design of long-lived parent processes, such as network servers and shells, that create numerous children. To put things another way, in such applications, a parent process should perform

wait() calls in order to ensure that dead children are always removed from the system, rather than becoming long-lived zombies. The parent may perform such *wait()* calls either synchronously, or asynchronously, in response to delivery of the SIGCHLD signal, as described in Section 26.3.1.

Listing 26-4 demonstrates the creation of a zombie and that a zombie can't be killed by SIGKILL. When we run this program, we see the following output:

```
$ ./make_zombie
Parent PID=1013
Child (PID=1014) exiting
 1013 pts/4    00:00:00 make_zombie                        Output from ps(1)
 1014 pts/4    00:00:00 make_zombie <defunct>
After sending SIGKILL to make_zombie (PID=1014):
 1013 pts/4    00:00:00 make_zombie                        Output from ps(1)
 1014 pts/4    00:00:00 make_zombie <defunct>
```

In the above output, we see that *ps(1)* displays the string <defunct> to indicate a process in the zombie state.

> The program in Listing 26-4 uses the *system()* function to execute the shell command given in its character-string argument. We describe *system()* in detail in Section 27.6.

Listing 26-4: Creating a zombie child process

procexec/make_zombie.c

```
#include <signal.h>
#include <libgen.h>             /* For basename() declaration */
#include "tlpi_hdr.h"

#define CMD_SIZE 200

int
main(int argc, char *argv[])
{
    char cmd[CMD_SIZE];
    pid_t childPid;

    setbuf(stdout, NULL);       /* Disable buffering of stdout */

    printf("Parent PID=%ld\n", (long) getpid());

    switch (childPid = fork()) {
    case -1:
        errExit("fork");

    case 0:     /* Child: immediately exits to become zombie */
        printf("Child (PID=%ld) exiting\n", (long) getpid());
        _exit(EXIT_SUCCESS);
```

```
    default:    /* Parent */
        sleep(3);               /* Give child a chance to start and exit */
        snprintf(cmd, CMD_SIZE, "ps | grep %s", basename(argv[0]));
        cmd[CMD_SIZE - 1] = '\0';       /* Ensure string is null-terminated */
        system(cmd);            /* View zombie child */

        /* Now send the "sure kill" signal to the zombie */

        if (kill(childPid, SIGKILL) == -1)
            errMsg("kill");
        sleep(3);               /* Give child a chance to react to signal */
        printf("After sending SIGKILL to zombie (PID=%ld):\n", (long) childPid);
        system(cmd);            /* View zombie child again */

        exit(EXIT_SUCCESS);
    }
}
```

procexec/make_zombie.c

26.3 The SIGCHLD Signal

The termination of a child process is an event that occurs asynchronously. A parent can't predict when one of its child will terminate. (Even if the parent sends a SIGKILL signal to the child, the exact time of termination is still dependent on when the child is next scheduled for use of a CPU.) We have already seen that the parent should use *wait()* (or similar) in order to prevent the accumulation of zombie children, and have looked at two ways in which this can be done:

- The parent can call *wait()*, or *waitpid()* without specifying the WNOHANG flag, in which case the call will block if a child has not already terminated.
- The parent can periodically perform a nonblocking check (a poll) for dead children via a call to *waitpid()* specifying the WNOHANG flag.

Both of these approaches can be inconvenient. On the one hand, we may not want the parent to be blocked waiting for a child to terminate. On the other hand, making repeated nonblocking *waitpid()* calls wastes CPU time and adds complexity to an application design. To get around these problems, we can employ a handler for the SIGCHLD signal.

26.3.1 Establishing a Handler for SIGCHLD

The SIGCHLD signal is sent to a parent process whenever one of its children terminates. By default, this signal is ignored, but we can catch it by installing a signal handler. Within the signal handler, we can use *wait()* (or similar) to reap the zombie child. However, there is a subtlety to consider in this approach.

In Sections 20.10 and 20.12, we observed that when a signal handler is called, the signal that caused its invocation is temporarily blocked (unless the *sigaction()* SA_NODEFER flag was specified), and also that standard signals, of which SIGCHLD is one, are not queued. Consequently, if a second and third child terminate in quick

succession while a SIGCHLD handler is executing for an already terminated child, then, although SIGCHLD is generated twice, it is queued only once to the parent. As a result, if the parent's SIGCHLD handler called *wait()* only once each time it was invoked, the handler might fail to reap some zombie children.

The solution is to loop inside the SIGCHLD handler, repeatedly calling *waitpid()* with the WNOHANG flag until there are no more dead children to be reaped. Often, the body of a SIGCHLD handler simply consists of the following code, which reaps any dead children without checking their status:

```
while (waitpid(-1, NULL, WNOHANG) > 0)
    continue;
```

The above loop continues until *waitpid()* returns either 0, indicating no more zombie children, or −1, indicating an error (probably ECHILD, meaning that there are no more children).

Design issues for SIGCHLD handlers

Suppose that, at the time we establish a handler for SIGCHLD, there is already a terminated child for this process. Does the kernel then immediately generate a SIGCHLD signal for the parent? SUSv3 leaves this point unspecified. Some System V–derived implementations do generate a SIGCHLD in these circumstances; other implementations, including Linux, do not. A portable application can make this difference invisible by establishing the SIGCHLD handler before creating any children. (This is usually the natural way of doing things, of course.)

A further point to consider is the issue of reentrancy. In Section 21.1.2, we noted that using a system call (e.g., *waitpid()*) from within a signal handler may change the value of the global variable *errno*. Such a change could interfere with attempts by the main program to explicitly set *errno* (see, for example, the discussion of *getpriority()* in Section 35.1) or check its value after a failed system call. For this reason, it is sometimes necessary to code a SIGCHLD handler to save *errno* in a local variable on entry to the handler, and then restore the *errno* value just prior to returning. An example of this is shown in Listing 26-5.

Example program

Listing 26-5 provides an example of a more complex SIGCHLD handler. This handler displays the process ID and wait status of each reaped child ①. In order to see that multiple SIGCHLD signals are not queued while the handler is already invoked, execution of the handler is artificially lengthened by a call to *sleep()* ②. The main program creates one child process for each (integer) command-line argument ④. Each of these children sleeps for the number of seconds specified in the corresponding command-line argument and then exits ⑤. In the following example of the execution of this program, we see that even though three children terminate, SIGCHLD is only queued twice to the parent:

```
$ ./multi_SIGCHLD 1 2 4
16:45:18 Child 1 (PID=17767) exiting
16:45:18 handler: Caught SIGCHLD            First invocation of handler
16:45:18 handler: Reaped child 17767 - child exited, status=0
```

```
16:45:19 Child 2 (PID=17768) exiting        These children terminate during...
16:45:21 Child 3 (PID=17769) exiting        first invocation of handler
16:45:23 handler: returning                 End of first invocation of handler
16:45:23 handler: Caught SIGCHLD            Second invocation of handler
16:45:23 handler: Reaped child 17768 - child exited, status=0
16:45:23 handler: Reaped child 17769 - child exited, status=0
16:45:28 handler: returning
16:45:28 All 3 children have terminated; SIGCHLD was caught 2 times
```

Note the use of *sigprocmask()* to block the SIGCHLD signal before any children are created in Listing 26-5 ③. This is done to ensure correct operation of the *sigsuspend()* loop in the parent. If we failed to block SIGCHLD in this way, and a child terminated between the test of the value of *numLiveChildren* and the execution of the *sigsuspend()* call (or alternatively a *pause()* call), then the *sigsuspend()* call would block forever waiting for a signal that has already been caught ⑥. The requirement for dealing with this type of race condition was detailed in Section 22.9.

Listing 26-5: Reaping dead children via a handler for SIGCHLD

procexec/multi_SIGCHLD.c

```
#include <signal.h>
#include <sys/wait.h>
#include "print_wait_status.h"
#include "curr_time.h"
#include "tlpi_hdr.h"

static volatile int numLiveChildren = 0;
                /* Number of children started but not yet waited on */

static void
sigchldHandler(int sig)
{
    int status, savedErrno;
    pid_t childPid;

    /* UNSAFE: This handler uses non-async-signal-safe functions
       (printf(), printWaitStatus(), currTime(); see Section 21.1.2) */

    savedErrno = errno;     /* In case we modify 'errno' */

    printf("%s handler: Caught SIGCHLD\n", currTime("%T"));

    while ((childPid = waitpid(-1, &status, WNOHANG)) > 0) {
①       printf("%s handler: Reaped child %ld - ", currTime("%T"),
                (long) childPid);
        printWaitStatus(NULL, status);
        numLiveChildren--;
    }

    if (childPid == -1 && errno != ECHILD)
        errMsg("waitpid");
```

```
②      sleep(5);               /* Artificially lengthen execution of handler */
       printf("%s handler: returning\n", currTime("%T"));

       errno = savedErrno;
   }

   int
   main(int argc, char *argv[])
   {
       int j, sigCnt;
       sigset_t blockMask, emptyMask;
       struct sigaction sa;

       if (argc < 2 || strcmp(argv[1], "--help") == 0)
           usageErr("%s child-sleep-time...\n", argv[0]);

       setbuf(stdout, NULL);       /* Disable buffering of stdout */

       sigCnt = 0;
       numLiveChildren = argc - 1;

       sigemptyset(&sa.sa_mask);
       sa.sa_flags = 0;
       sa.sa_handler = sigchldHandler;
       if (sigaction(SIGCHLD, &sa, NULL) == -1)
           errExit("sigaction");

       /* Block SIGCHLD to prevent its delivery if a child terminates
          before the parent commences the sigsuspend() loop below */

       sigemptyset(&blockMask);
       sigaddset(&blockMask, SIGCHLD);
③      if (sigprocmask(SIG_SETMASK, &blockMask, NULL) == -1)
           errExit("sigprocmask");

④      for (j = 1; j < argc; j++) {
           switch (fork()) {
           case -1:
               errExit("fork");

           case 0:         /* Child - sleeps and then exits */
⑤              sleep(getInt(argv[j], GN_NONNEG, "child-sleep-time"));
               printf("%s Child %d (PID=%ld) exiting\n", currTime("%T"),
                       j, (long) getpid());
               _exit(EXIT_SUCCESS);

           default:        /* Parent - loops to create next child */
               break;
           }
       }
```

```
    /* Parent comes here: wait for SIGCHLD until all children are dead */

    sigemptyset(&emptyMask);
    while (numLiveChildren > 0) {
⑥       if (sigsuspend(&emptyMask) == -1 && errno != EINTR)
            errExit("sigsuspend");
        sigCnt++;
    }

    printf("%s All %d children have terminated; SIGCHLD was caught "
            "%d times\n", currTime("%T"), argc - 1, sigCnt);

    exit(EXIT_SUCCESS);
}
```

procexec/multi_SIGCHLD.c

26.3.2 Delivery of SIGCHLD for Stopped Children

Just as *waitpid()* can be used to monitor stopped children, so is it possible for a parent process to receive the SIGCHLD signal when one of its children is stopped by a signal. This behavior is controlled by the SA_NOCLDSTOP flag when using *sigaction()* to establish a handler for the SIGCHLD signal. If this flag is omitted, a SIGCHLD signal is delivered to the parent when one of its children stops; if the flag is present, SIGCHLD is not delivered for stopped children. (The implementation of *signal()* given in Section 22.7 doesn't specify SA_NOCLDSTOP.)

> Since SIGCHLD is ignored by default, the SA_NOCLDSTOP flag has a meaning only if we are establishing a handler for SIGCHLD. Furthermore, SIGCHLD is the only signal for which the SA_NOCLDSTOP flag has an effect.

SUSv3 also allows for a parent to be sent a SIGCHLD signal if one of its stopped children is resumed by being sent a SIGCONT signal. (This corresponds to the WCONTINUED flag for *waitpid()*.) This feature is implemented in Linux since kernel 2.6.9.

26.3.3 Ignoring Dead Child Processes

There is a further possibility for dealing with dead child processes. Explicitly setting the disposition of SIGCHLD to SIG_IGN causes any child process that subsequently terminates to be immediately removed from the system instead of being converted into a zombie. In this case, since the status of the child process is simply discarded, a subsequent call to *wait()* (or similar) can't return any information for the terminated child.

> Note that even though the default disposition for SIGCHLD is to be ignored, explicitly setting the disposition to SIG_IGN causes the different behavior described here. In this respect, SIGCHLD is treated uniquely among signals.

On Linux, as on many UNIX implementations, setting the disposition of SIGCHLD to SIG_IGN doesn't affect the status of any existing zombie children, which must still be waited upon in the usual way. On some other UNIX implementations (e.g., Solaris 8), setting the disposition of SIGCHLD to SIG_IGN *does* remove existing zombie children.

The SIG_IGN semantics for SIGCHLD have a long history, deriving from System V. SUSv3 specifies the behavior described here, but these semantics were left unspecified in the original POSIX.1 standard. Thus, on some older UNIX implementations, ignoring SIGCHLD has no effect on the creation of zombies. The only completely portable way of preventing the creation of zombies is to call *wait()* or *waitpid()*, possibly from within a handler established for SIGCHLD.

Deviations from SUSv3 in older Linux kernels

SUSv3 specifies that if the disposition of SIGCHLD is set to SIG_IGN, the resource usage information for the child should be discarded and not included in the totals returned when the parent makes a call to *getrusage()* specifying the RUSAGE_CHILDREN flag (Section 36.1). However, on Linux versions before kernel 2.6.9, the CPU times and resources used by the child *are* recorded and are visible in calls to *getrusage()*. This nonconformance is fixed in Linux 2.6.9 and later.

> Setting the disposition of SIGCHLD to SIG_IGN should also prevent the child CPU times from being included in the structure returned by *times()* (Section 10.7). However, on Linux kernels before 2.6.9, a similar nonconformance applies for the information returned by *times()*.

SUSv3 specifies that if the disposition of SIGCHLD is set to SIG_IGN, and the parent has no terminated children that have been transformed into zombies and have not yet been waited for, then a call to *wait()* (or *waitpid()*) should block until *all* of the parent's children have terminated, at which point the call should terminate with the error ECHILD. Linux 2.6 conforms to this requirement. However, in Linux 2.4 and earlier, *wait()* blocks only until the *next* child terminates, and then returns the process ID and status of that child (i.e., the behavior is the same as if the disposition of SIGCHLD had not been set to SIG_IGN).

The *sigaction()* SA_NOCLDWAIT flag

SUSv3 specifies the SA_NOCLDWAIT flag, which can be used when setting the disposition of the SIGCHLD signal using *sigaction()*. This flag produces behavior similar to that when the disposition of SIGCHLD is set to SIG_IGN. This flag was not implemented in Linux 2.4 and earlier, but is implemented in Linux 2.6.

The principal difference between setting the disposition of SIGCHLD to SIG_IGN and employing SA_NOCLDWAIT is that, when establishing a handler with SA_NOCLDWAIT, SUSv3 leaves it unspecified whether or not a SIGCHLD signal is sent to the parent when a child terminates. In other words, an implementation is permitted to deliver SIGCHLD when SA_NOCLDWAIT is specified, and an application could catch this signal (although the SIGCHLD handler would not be able to reap the child status using *wait()*, since the kernel has already discarded the zombie). On some UNIX implementations, including Linux, the kernel does generate a SIGCHLD signal for the parent process. On other UNIX implementations, SIGCHLD is not generated.

> When setting the SA_NOCLDWAIT flag for the SIGCHLD signal, older Linux kernels demonstrate the same details of nonconformance to SUSv3 as were described above for setting the disposition of SIGCHLD to SIG_IGN.

The System V SIGCLD signal

On Linux, the name SIGCLD is provided as a synonym for the SIGCHLD signal. The reason for the existence of both names is historical. The SIGCHLD signal originated on BSD, and this name was adopted by POSIX, which largely standardized on the BSD signal model. System V provided the corresponding SIGCLD signal, with slightly different semantics.

The key difference between BSD SIGCHLD and System V SIGCLD lies in what happens when the disposition of the signal was set to SIG_IGN:

- On historical (and some contemporary) BSD implementations, the system continues to generate zombies for unwaited-for children, even when SIGCHLD is ignored.
- On System V, using *signal()* (but not *sigaction()*) to ignore SIGCLD has the result that zombies are not generated when children died.

As already noted, the original POSIX.1 standard left the result of ignoring SIGCHLD unspecified, thus permitting the System V behavior. Nowadays, this System V behavior is specified as part of SUSv3 (which nevertheless holds to the name SIGCHLD). Modern System V derivatives use the standard name SIGCHLD for this signal, but continue to provide the synonym SIGCLD. Further details on SIGCLD can be found in [Stevens & Rago, 2005].

26.4 Summary

Using *wait()* and *waitpid()* (and other related functions), a parent process can obtain the status of its terminated and stopped children. This status indicates whether a child process terminated normally (with an exit status indicating either success or failure), terminated abnormally, was stopped by a signal, or was resumed by a SIGCONT signal.

If a child's parent terminates, the child becomes an orphan and is adopted by the *init* process, whose process ID is 1.

When a child process terminates, it becomes a zombie, and is removed from the system only when its parent calls *wait()* (or similar) to retrieve the child's status. Long-running programs such as shells and daemons should be designed so that they always reap the status of the child processes they create, since a process in the zombie state can't be killed, and unreaped zombies will eventually clog the kernel process table.

A common way of reaping dead child processes is to establish a handler for the SIGCHLD signal. This signal is delivered to a parent process whenever one of its children terminates, and optionally when a child is stopped by a signal. Alternatively, but somewhat less portably, a process may elect to set the disposition of SIGCHLD to SIG_IGN, in which case the status of terminated children is immediately discarded (and thus can't later be retrieved by the parent), and the children don't become zombies.

Further information

Refer to the sources of further information listed in Section 24.6.

26.5 Exercises

26-1. Write a program to verify that when a child's parent terminates, a call to *getppid()* returns 1 (the process ID of *init*).

26-2. Suppose that we have three processes related as grandparent, parent, and child, and that the grandparent doesn't immediately perform a *wait()* after the parent exits, so that the parent becomes a zombie. When do you expect the grandchild to be adopted by *init* (so that *getppid()* in the grandchild returns 1): after the parent terminates or after the grandparent does a *wait()*? Write a program to verify your answer.

26-3. Replace the use of *waitpid()* with *waitid()* in Listing 26-3 (`child_status.c`). The call to *printWaitStatus()* will need to be replaced by code that prints relevant fields from the *siginfo_t* structure returned by *waitid()*.

26-4. Listing 26-4 (`make_zombie.c`) uses a call to *sleep()* to allow the child process a chance to execute and terminate before the parent executes *system()*. This approach produces a theoretical race condition. Modify the program to eliminate the race condition by using signals to synchronize the parent and child.