Next Previous Contents

10. Processes

Before looking at the Linux implementation, first a general Unix description of threads, processes, process groups and sessions.

A session contains a number of process groups, and a process group contains a number of processes, and a process contains a number of threads.

A session can have a controlling tty. At most one process group in a session can be a foreground process group. An interrupt character typed on a tty ("Teletype", i.e., terminal) causes a signal to be sent to all members of the foreground process group in the session (if any) that has that tty as controlling tty.

All these objects have numbers, and we have thread IDs, process IDs, process group IDs and session IDs.

10.1 Processes

Creation

A new process is traditionally started using the `fork()` system call:

```
pid_t p;

p = fork();
if (p == (pid_t) -1)
        /* ERROR */
else if (p == 0)
        /* CHILD */
else
        /* PARENT */
```

This creates a child as a duplicate of its parent. Parent and child are identical in almost all respects. In the code they are distinguished by the fact that the parent learns the process ID of its child, while `fork()` returns 0 in the child. (It can find the process ID of its parent using the `getppid()` system call.)

Termination

Normal termination is when the process does

```
exit(n);
```

or

```
return n;
```

from its `main()` procedure. It returns the single byte `n` to its parent.

Abnormal termination is usually caused by a signal.

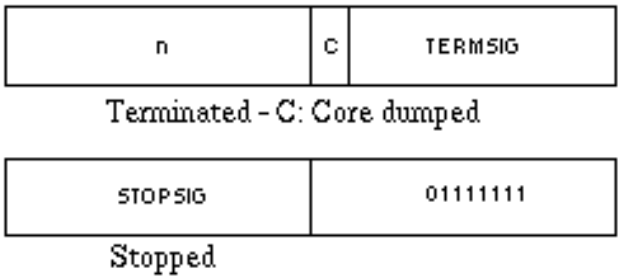
Collecting the exit code. Zombies

The parent does

```
pid_t p;
int status;

p = wait(&status);
```

and collects two bytes:



A process that has terminated but has not yet been waited for is a *zombie*. It need only store these two bytes: exit code and reason for termination.

On the other hand, if the parent dies first, `init` (process 1) inherits the child and becomes its parent.

Signals

Stopping

Some signals cause a process to stop: `SIGSTOP` (stop!), `SIGTSTP` (stop from tty: probably ^Z was typed), `SIGTTIN` (tty input asked by background process), `SIGTTOU` (tty output sent by background process, and this was disallowed by `stty tostop`).

Apart from ^Z there also is ^Y. The former stops the process when it is typed, the latter stops it when it is read.

Signals generated by typing the corresponding character on some tty are sent to all processes that are in the foreground process group of the session that has that tty as controlling tty. (Details below.)

If a process is being traced, every signal will stop it.

Continuing

`SIGCONT`: continue a stopped process.

Terminating

`SIGKILL` (die! now!), `SIGTERM` (please, go away), `SIGHUP` (modem hangup), `SIGINT` (^C), `SIGQUIT` (^\), etc. Many signals have as default action to kill the target. (Sometimes with an additional core dump, when such is allowed by rlimit.) The signals `SIGCHLD` and `SIGWINCH` are ignored by default. All except `SIGKILL` and `SIGSTOP` can be caught or ignored or blocked. For details, see `signal(7)`.

10.2 Process groups

Every process is member of a unique *process group*, identified by its *process group ID*. (When the process is created, it becomes a member of the process group of its parent.) By convention, the process group ID of a process group equals the process ID of the first member of the process group, called the *process group leader*. A process finds the ID of its process group using the system call `getpgrp()`, or, equivalently, `getpgid(0)`. One finds the process group ID of process `p` using `getpgid(p)`.

One may use the command `ps j` to see PPID (parent process ID), PID (process ID), PGID (process group ID) and SID (session ID) of processes. With a shell that does not know about job control, like `ash`, each of its children will be in the same session and have the same process group as the shell. With a shell that knows about job control, like `bash`, the processes of one pipeline. like

```
% cat paper | ideal | pic | tbl | eqn | ditroff > out
```

form a single process group.

Creation

A process `pid` is put into the process group `pgid` by

```
setpgid(pid, pgid);
```

If `pgid == pid` or `pgid == 0` then this creates a new process group with process group leader `pid`. Otherwise, this puts `pid` into the already existing process group `pgid`. A zero `pid` refers to the current process. The call `setpgrp()` is equivalent to `setpgid(0,0)`.

Restrictions on setpgid()

The calling process must be `pid` itself, or its parent, and the parent can only do this before `pid` has done `exec()`, and only when both belong to the same session. It is an error if process `pid` is a session leader (and this call would change its `pgid`).

Typical sequence

```
p = fork();
if (p == (pid_t) -1) {
        /* ERROR */
} else if (p == 0) {    /* CHILD */
        setpgid(0, pgid);
        ...
} else {                /* PARENT */
        setpgid(p, pgid);
        ...
}
```

This ensures that regardless of whether parent or child is scheduled first, the process group setting is as expected by both.

Signalling and waiting

One can signal all members of a process group:

```
killpg(pgrp, sig);
```

One can wait for children in ones own process group:

```
waitpid(0, &status, ...);
```

or in a specified process group:

```
waitpid(-pgrp, &status, ...);
```

Foreground process group

Among the process groups in a session at most one can be the *foreground process group* of that session. The tty input and tty signals (signals generated by ^C, ^Z, etc.) go to processes in this foreground process group.

A process can determine the foreground process group in its session using `tcgetpgrp(fd)`, where `fd` refers to its controlling tty. If there is none, this returns a random value larger than 1 that is not a process group ID.

A process can set the foreground process group in its session using `tcsetpgrp(fd,pgrp)`, where `fd` refers to its controlling tty, and `pgrp` is a process group in the its session, and this session still is associated to the controlling tty of the calling process.

How does one get `fd`? By definition, `/dev/tty` refers to the controlling tty, entirely independent of redirects of standard input and output. (There is also the function `ctermid()` to get the name of the controlling terminal. On a POSIX standard system it will return `/dev/tty`.) Opening the name of the controlling tty gives a file descriptor `fd`.

Background process groups

All process groups in a session that are not foreground process group are *background process groups*. Since the user at the keyboard is interacting with foreground processes, background processes should stay away from it. When a background process reads from the terminal it gets a SIGTTIN signal. Normally, that will stop it, the job control shell notices and tells the user, who can say `fg` to continue this background process as a foreground process, and then this process can read from the terminal. But if the background process ignores or blocks the SIGTTIN signal, or if its process group is orphaned (see below), then the read() returns an EIO error, and no signal is sent. (Indeed, the idea is to tell the process that reading from the terminal is not allowed right now. If it wouldn't see the signal, then it will see the error return.)

When a background process writes to the terminal, it may get a SIGTTOU signal. May: namely, when the flag that this must happen is set (it is off by default). One can set the flag by

```
% stty tostop
```

and clear it again by

```
% stty -tostop
```

and inspect it by

```
% stty -a
```

Again, if TOSTOP is set but the background process ignores or blocks the SIGTTOU signal, or if its process group is orphaned (see below), then the write() returns an EIO error, and no signal is sent.

Orphaned process groups

The process group leader is the first member of the process group. It may terminate before the others, and then the process group is without leader.

A process group is called *orphaned* when *the parent of every member is either in the process group or outside the session*. In particular, the process group of the session leader is always orphaned.

If termination of a process causes a process group to become orphaned, and some member is stopped, then all are sent first SIGHUP and then SIGCONT.

The idea is that perhaps the parent of the process group leader is a job control shell. (In the same session but a different process group.) As long as this parent is alive, it can handle the stopping and starting of members in the process group. When it dies, there may be nobody to continue stopped processes. Therefore, these stopped processes are sent SIGHUP, so that they die unless they catch or ignore it, and then SIGCONT to continue them.

Note that the process group of the session leader is already orphaned, so no signals are sent when the session leader dies.

Note also that a process group can become orphaned in two ways by termination of a process: either it was a parent and not itself in the process group, or it was the last element of the process group with a parent outside but in the same session. Furthermore, that a process group can become orphaned other than by termination of a process, namely when some member is moved to a different process group.

10.3 Sessions

Every process group is in a unique *session*. (When the process is created, it becomes a member of the session of its parent.) By convention, the session ID of a session equals the process ID of the first member of the session, called the *session leader*. A process finds the ID of its session using the system call `getsid()`.

Every session may have a *controlling tty*, that then also is called the controlling tty of each of its member processes. A file descriptor for the controlling tty is obtained by opening `/dev/tty`. (And when that fails, there was no controlling tty.) Given a file descriptor for the controlling tty, one may obtain the SID using `tcgetsid(fd)`.

A session is often set up by a login process. The terminal on which one is logged in then becomes the controlling tty of the session. All processes that are descendants of the login process will in general be members of the session.

Creation

A new session is created by

```
pid = setsid();
```

This is allowed only when the current process is not a process group leader. In order to be sure of that we fork first:

```
p = fork();
if (p) exit(0);
pid = setsid();
```

The result is that the current process (with process ID `pid`) becomes session leader of a new session with session ID `pid`. Moreover, it becomes process group leader of a new process group. Both session and process group contain only the single process `pid`. Furthermore, this process has no controlling tty.

The restriction that the current process must not be a process group leader is needed: otherwise its PID serves as PGID of some existing process group and cannot be used as the PGID of a new process group.

Getting a controlling tty

How does one get a controlling terminal? Nobody knows, this is a great mystery.

The System V approach is that the first tty opened by the process becomes its controlling tty.

The BSD approach is that one has to explicitly call

```
ioctl(fd, TIOCSCTTY, ...);
```

to get a controlling tty.

Linux tries to be compatible with both, as always, and this results in a very obscure complex of conditions. Roughly:

The `TIOCSCTTY` ioctl will give us a controlling tty, provided that (i) the current process is a session leader, and (ii) it does not yet have a controlling tty, and (iii) maybe the tty should not already control some other session; if it does it is an error if we aren't root, or we steal the tty if we are all-powerful.

Opening some terminal will give us a controlling tty, provided that (i) the current process is a session leader, and (ii) it does not yet have a controlling tty, and (iii) the tty does not already control some other session, and (iv) the open did not have the `O_NOCTTY` flag, and (v) the tty is not the foreground VT, and (vi) the tty is not the console, and (vii) maybe the tty should not be master or slave pty.

Getting rid of a controlling tty

If a process wants to continue as a daemon, it must detach itself from its controlling tty. Above we saw that `setsid()` will remove the controlling tty. Also the ioctl TIOCNOTTY does this. Moreover, in order not to get a controlling tty again as soon as it opens a tty, the process has to fork once more, to assure that it is not a session leader. Typical code fragment:

```
if ((fork()) != 0)
        exit(0);
setsid();
if ((fork()) != 0)
        exit(0);
```

See also `daemon(3)`.

Disconnect

If the terminal goes away by modem hangup, and the line was not local, then a SIGHUP is sent to the session leader. Any further reads from the gone terminal return EOF. (Or possibly -1 with `errno` set to EIO.)

If the terminal is the slave side of a pseudotty, and the master side is closed (for the last time), then a SIGHUP is sent to the foreground process group of the slave side.

When the session leader dies, a SIGHUP is sent to all processes in the foreground process group. Moreover, the terminal stops being the controlling terminal of this session (so that it can become the controlling terminal of another session).

Thus, if the terminal goes away and the session leader is a job control shell, then it can handle things for its descendants, e.g. by sending them again a SIGHUP. If on the other hand the session leader is an innocent process that does not catch SIGHUP, it will die, and all foreground processes get a SIGHUP.

10.4 Threads

A process can have several threads. New threads (with the same PID as the parent thread) are started using the `clone` system call using the `CLONE_THREAD` flag. Threads are distinguished by a *thread ID* (TID). An ordinary process has a single thread with TID equal to PID. The system call `gettid()` returns the TID. The system call `tkill()` sends a signal to a single thread.

Example: a process with two threads. Both only print PID and TID and exit. (Linux 2.4.19 or later.)

```
% cat << EOF > gettid-demo.c
#include <unistd.h>
#include <sys/types.h>
#define CLONE_SIGHAND   0x00000800
#define CLONE_THREAD    0x00010000
#include <linux/unistd.h>
#include <errno.h>
_syscall0(pid_t,gettid)

int thread(void *p) {
        printf("thread: %d %d\n", gettid(), getpid());
}

main() {
        unsigned char stack[4096];
        int i;

        i = clone(thread, stack+2048, CLONE_THREAD | CLONE_SIGHAND, NULL);
        if (i == -1)
                perror("clone");
        else
                printf("clone returns %d\n", i);
        printf("parent: %d %d\n", gettid(), getpid());
}
EOF
% cc -o gettid-demo gettid-demo.c
% ./gettid-demo
clone returns 21826
parent: 21825 21825
thread: 21826 21825
%
```

Next Previous Contents