Operating Systems
COMS W4118
Reading Notes 4

Alexander Roth

2015 – 02 – 19

Advanced Programming in the Unix Environment Chapter 9: Process Relationships

9.4 Process Groups

- A process group is a collection of one or more processes, usually associated with the same job, that can receive signals from the same terminal.
- Each process group has a unique process group ID.
- The function `getpgrp` returns the process group ID of the calling process.

```
#include <unistd.h>
pid_t getpgrp(void);
```
Returns: process group ID of calling process.

```
#include <unistd.h>
pid_t getpgid(pid_t pid);
```
Returns: process group ID if OK, -1 on error.

- Each process group can have a process group leader. The leader is identified by its process group ID being equal to its process ID.
- The process group lifetime is the period of time that begins when a group is created and ends when the last remaining process leaves the group.
- A process joins an existing process group or creates a new process group by calling `setpgid`.

```
#include <unistd.h>
int setpgid(pid_t pid, pid_t pgid);
```
Returns: 0 if OK, -1 on error.

- Sets the process group ID to *pgid* in the process whose process ID equals *pid*.
- If the two arguments are equal, the process specified becomes a process group leader.
- A process can set the process group ID of only itself or any of its children.
- This function is often called after a `fork` by the parent process to set the child's process group ID and by the child to set it's own child group ID. We have the redundancy to avoid any race conditions.

9.5 Sessions

- A session is a collection of one or more process groups.
- The processes in a process group are usually placed into their respective process group by a shell pipeline.
- A process establishes a new session by calling the `setsid` function.

```
#include <unistd.h>
pid_t setsid(void);
```
Returns: process group ID if OK, -1 on error

- If the calling process is not a process group leader, this function creates a new session.
- Three things happen afterwards:
    1. The process becomes the *session leader* and is the only process in this new session. A session leader is the process that creates a session.
    2. The process becomes the process group leader of a new procress group; the process ID of the calling process is the process group ID.
    3. The process has no controlling terminal.
- This function returns an error if the caller is already a process group leader.
- The `getsid` function returns the process group ID of a process's session leader.
- The session leader is always the leader of a process group.

```
#include <unistd.h>
pid_t getsid(pid_t pid);
```
Returns: session leader's process group ID if OK, -1 on error

9.6 Controlling Terminal

- A session can have a single *controlling terminal*, which is usually the terminal device or pseudo terminal device on which we log in.
- The session leader that establishes the connection to the controlling terminal is called the *controlling process*.
- The proces groups within a session can be divided into a single *foreground process group* and one or more *background process groups*.
- If a session has a controlling terminal, it has a single foreground proces group and all other process groups in the session are background process groups.
- Whenever we press the terminal's interrupt key, the interrupt signal is sent to all processes in the foreground process group.
- Whenever we press the terminal's quit key, the quit signal is sent to all processes in the foreground process group.
- If a modem disconnect is detected by the terminal interface, the hang-up signal is sent to the controlling process (session leader).
- A program guarantees it is talking ot the controlling terminal when it opens the file `/dev/tty`. This special file is synonomous within the kernel for the controlling terminal.

Chapter 10: Signals

10.8 Reliable-Signal Terminology and Semantics

- A signal is *generate* for a process when the event that causes the signal occurs. The event could be a hardware exception, a software condition, or anything else.
- A signal is *delivered* to a process when the action for a signal is taken.
- During the time between the generation of a signal and its delivery, the signal is said to be *pending*.
- A process has the option of *blocking* the delivery of a signal.
- If a signal that is blocked is generated for a process, and if the action for that signal is either the default action or to catch the signal, then the signal remains pending for the process until the process either unblocks the signal or changes the action to ignore the signal.
- If the system delivers the signal more than once, we say that the signals are queued.

- Most UNIX systems do not queue signals. The kernel simply delivers the signal once.
- POSIX.1 does not specify the order in which the signals are delivered to the process. Signals related to the current state of the process should be delivered before other signals.
- Each process has a *signal mask* that defines the set of signals currently blocked from delivery to that process.

0.1 10.9 `kill` and `raise` Functions

- The `kill` function sends a signal to a process or a group of processes.
- The `raise` function allows a process to send a signal to itself.

```
#include <signal.h>
int kill(pid_t pid, int signo);
int raise(int signo);
```
Both return: 0 if OK, -1 on error

- There are four different conditions for the *pid* argument to `kill`.

  $pid > 0$ The signal is sent to the process whose process ID is *pid*.

  $pid == 0$ The signal is sent to all processes whose process group ID equals the process group ID of the sender and for which the sender has permission to send the signal.

  $pid < 0$ The signal is sent to all processes whose process group ID equals the absolute value of *pid* and for which the sender has permission to send the signal.

  $pid == -1$ The signal is sent to all processes on the system for which the sender has permission to send the signal.
- A process needs permission to send a signal to another process.
- The superuser can send a signal to any process.
- POSIX.1 defines signal number 0 as the null signal.
- If the *signo* argument is 0, then the normal error checking is performed by `kill`, but no signal is sent.
- The call to `kill` is not atomic.

10.10 `alarm` and `pause` Functions

- The `alarm` function allows us to set a timer that will expire at a specified time in the future.
- When the timer expires, the `SIGALRM` signal is generated.
- The default action with `SIGALRM` is to terminate the process.

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```
Returns: 0 or number of seconds until previously set alarm

- The *seconds* value is the number of clock seconds in the future when the signal should be generated.
- If we intend to catch `SIGALRM`, we need to install its signal handler before calling `alarm`.
- The `pause` function suspends the calling process until a signal is caught.

```
#include <unistd.h>
int pause(void);
```
Returns: -1 with `errno` set to `EINTR`.

- Only returns when a signal handler is executed and returns.
- Returns with `-1` with `errno` set to `EINTR`.

10.11 Signal Sets

- POSIX.1 defines the data type `sigset_t` to contain a signal set and the following five functions to manipulate singal sets.

```
#include <signal.h>
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
```
All four return: 0 if OK, -1 on error

```
int sigismember(const sigset_t *set, int signo);
```
Returns: 1 if true, 0 if false, -1 on error.

- `sigemptyset` initializes the signal set pointed to by *set* so that all signals are excluded.
- `sigfillset` initializes the signal set so that all signals are included.
- Once we have initialized a signal set, we can add and delete specific signals in the set.

- `sigaddset` adds a single signal to an existing set.
- `sigdelset` removes a single signal from a set.
- All functions that take a signal set as an argument, we pass the address of the signal set as the argument.

10.12 `sigprocmask` Function

- A process can examine its signal mask, change its signal mask, or perform both operations in one step by calling the following function.

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *restrict set, sigset_t *restrict
oset);
```

Returns: 0 if OK, -1 on error

- If *oset* is a non-null pointer, the current signal mask for the process is returned through *oset*.
- If *set* is a non-null pointer, the *how* argument indicates how the current signal mask is modified.
- If *set* is a null pointer, signal mask of the process is not changed, and *how* is ignored.

10.13 `sigpending` Function

- The `sigpending` function returns the set of signals that are blocked from delivery and currently pending for the calling process.

```
#include <signal.h>
int sigpending(sigset_t *set);
```

Returns: 0 if OK, -1 on error.

10.14 `sigaction` Function

- The `sigaction` function allows us to examine and/or modify the action associated with a particular signal.

```
#include <signal.h>
int sigaction(int signo, const struct sigaction *restrict act, struct
sigaction *restrict oact);
```

Returns: 0 if OK, -1 on error

- *signo* is the signal number whose action we are examining or modifying.
- If *act* is non-null, we are modifying the action.
- If *oact* is non-null, the system returns the previous action for the signal through the *oact* pointer.