CHAPTER 10

シグナル

10.1 はじめに

シグナルはソフトウェアによる割り込みである。単純なアプリケーションを除いて、ほとんどのプログラムではシグナルを扱う必要がある。シグナルは非同期な事象を扱う方法を提供する。例えば、プログラムを休止させるためにユーザが端末で割り込みキーを打ったり、パイプラインの後続のプログラムが先に終了した場合などである。

UNIX の初期のバージョンからシグナルは存在したが、バージョン 7 のようなシステムのシグナルモデルには信頼性がなかった。シグナルを紛失することもあり、プログラムの臨界領域を実行中に特定のシグナルを無効にすることが困難であった。4.3BSD と SVR3 では、シグナルモデルを変更し、信頼性のあるシグナルを追加した。しかし、バークレーと AT&T における変更方法の間に互換性はなかった。幸い、POSIX.1 では信頼できるシグナルルーティンを規定した。本章では、これについて述べる。

まず、シグナルを概観し、各シグナルの一般的な使用方法を述べる。続いて、初期の実装の問題について述べる。正しく行う方法を見る前に、実装の問題点を理解することがしばしば重要になってくる。本章には完全ではない例が数多くあり、それらの問題点についても述べる。

10.2 シグナルの概念

各シグナルには名前が付いている。これらの名前はすべて 3 つの文字 SIG で始まる。例えば、SIGABRT は、プロセスが abort 関数を呼ぶと生成されるアボート (異常終了) シグナルであ

る。SIGALRM は、alarm 関数で設定したタイマーが切れると生成されるアラームシグナルである。

バージョン 7 には 15 のシグナルがあり、SVR4 と 4.3+BSD には 31 のシグナルがある。

これらの名前すべてに、正の整数 (シグナル番号) がヘッダー <signal.h> で定義されている。シグナル番号が 0 のシグナルは存在しない。10.9 節では、kill 関数がシグナル番号 0 を特別扱いすることを見る。POSIX.1 では、この値を null シグナルと呼ぶ。

いろいろな状況でシグナルが生成される。

- ユーザが特別な端末キーを打つと、シグナルが生成される。端末の DELETE キーを打つと、通常、割り込みシグナル (SIGINT) が生成される。これは暴走しているプログラムを止める方法である。(第 11 章では、端末の任意のキーにこのシグナルを結び付ける方法を述べる。)
- ハードウェア例外 (0 除算、不正なメモリ参照など) によってシグナルが生成される。これらの状況は、通常、ハードウェアが検出してカーネルに通知する。カーネルは、これらの状況が発生したときに実行していたプロセスに対して適切なシグナルを生成する。例えば、不正なメモリ参照を行ったプロセスに対しては SIGSEGV を生成する。
- kill(2) 関数により、別のプロセスやプロセスグループに対して任意のシグナルを送ることができる。普通は制約があり、シグナルを送る先のプロセスの所有者であるか、スーパーユーザである必要がある。
- kill(1) コマンドにより、別のプロセスにシグナルを送ることができる。このプログラムは、kill 関数に対するインタフェースである。このコマンドは、暴走しているバックグラウンドプロセスを終了するためにしばしば使用する。
- プロセスに通知すべき事象が発生すると、ソフトウェア的にシグナルを発生する。これらは (0 除算のように) ハードウェアで生成されるものではなく、ソフトウェア上で生成される。例として は、SIGURG (ネットワークを介して緊急データが到着すると生成される)、SIGPIPE (パイプを 読み取るプロセスが終了しているにもかかわらずパイプへ書き込むと生成される)、SIGALRM (プロセスが設定したアラーム時計が経過した場合に生成される) がある。

シグナルは、非同期事象の古典的な例である。プロセスに対してランダムに発生する。プロセスで (errno のような) 変数を検査してシグナルの発生を調べることはできず、代わりに、カーネルに対して「このシグナルが発生したらこれを実行せよ」と指定する必要がある。

カーネルに対して指定するシグナル発生時に実行すべきことは、3 種類ある。これらを、シグナルの処置 (disposition) とかシグナルに結び付けられた動作 (action) と呼ぶ。

1. シグナルを無視する。ほとんどのシグナルに対してこの処置を適用できるが、2 つのシグナル SIGKILL と SIGSTOP だけは、無視できない。これら 2 つのシグナルを無視できない理由は、意のプロセスを強制終了したり休止する確実な手段をスーパーユーザに与えるためである。た、(不正なメモリ参照や 0 除算のような) ハードウェア例外により生成されたシグナルを ると、プロセスの挙動は未定義になる。
2. シグナルを捕捉する。この処置を行うには、シグナル発生時に呼び出すべき独自の関数をルに指示しておく。独自に用意した関数で、発生した状況を処理するために何を行って 例えば、コマンドインタープリタを書く場合、ユーザが端末で割り込みシグナルを発生 きには、実行中のコマンドを終了させて、プログラムのメインループへ戻る。SIGCHLD シグナルを捕捉した場合、子プロセスが終了したことを意味するため、シグナル捕捉関数では waitpid を呼び出して子のプロセス ID と終了状態を取得する。別の例として、一時ファイルを作成するプロセスでは、(kill コマンドがデフォルトで生成する終了シグナルである) SIGTERM シグナルを処理する関数で一時ファイルの後始末をする。
3. デフォルトの動作を適用する。各シグナルには図 10.1 に示すデフォルトの動作が定義されている。ほとんどのシグナルに対するデフォルト動作は、プロセスを終了させることになっていることに注意してほしい。

図 10.1 に、すべてのシグナルの名称、各システムでの使用の可否、シグナルに対するデフォルトの動作を示す。

■図 10.1 UNIX シグナル

名称	意味	ANSI C	POSIX.1	SVR4	4.3+BSD	デフォルト動作
SIGABRT	異常終了 (abort)	•	•	•	•	core を作成して終了
SIGALRM	タイムアウト (alarm)		•	•	•	終了
SIGBUS	ハードウェアフォルト			•	•	core を作成して終了
SIGCHLD	子の状態の変化		job	•	•	無視
SIGCONT	休止プロセスの再開		job	•	•	続行/無視
SIGEMT	ハードウェアフォルト			•	•	core を作成して終了
SIGFPE	算術演算例外	•	•	•	•	core を作成して終了
SIGHUP	ハングアップ		•	•	•	終了
SIGILL	不正なハードウェア命令	•	•	•	•	core を作成して終了
SIGINFO	キーボードからの状態報告要求				•	無視
SIGINT	端末の割り込み文字	•	•	•	•	終了
SIGIO	非同期入出力			•	•	終了/無視
SIGIOT	ハードウェアフォルト			•	•	core を作成して終了
SIGKILL	終了		•	•	•	終了
SIGPIPE	読み手のいないパイプへの書き出し		•	•	•	終了
SIGPOLL	ポーリング事象 (poll)			•		終了
SIGPROF	プロフィール時計アラーム (setitimer)			•	•	終了
SIGPWR	電源異常/リスタート			•		無視
SIGQUIT	端末のクイット文字		•	•	•	core を作成して終了
SIGSEGV	不正なメモリ参照	•	•	•	•	core を作成して終了
SIGSTOP	休止		job	•	•	プロセスを休止する
SIGSYS	不正なシステムコール			•	•	core を作成して終了
SIGTERM	終了	•	•	•	•	終了
SIGTRAP	ハードウェアフォルト			•	•	core を作成して終了
SIGTSTP	端末のサスペンド文字		job	•	•	プロセスを休止する
SIGTTIN	バックグラウンドが制御端末から入力		job	•	•	プロセスを休止する
SIGTTOU	バックグラウンドが制御端末へ出力		job	•	•	プロセスを休止する
SIGURG	緊急状態			•	•	無視
SIGUSR1	ユーザ定義シグナル		•	•	•	終了
SIGUSR2	ユーザ定義シグナル		•	•	•	終了
SIGVTALRM	仮想時計アラーム (setitimer)			•	•	終了
SIGWINCH	端末のウィンドウサイズの変更			•	•	無視
SIGXCPU	CPU リミットの超過 (setrlimit)			•	•	core を作成して終了
SIGXFSZ	ファイルサイズリミットの超過 (setrlimit)			•	•	core を作成して終了

1 の欄では、必須のシグナルには • 印を、(ジョブ制御を扱う場合には必須の) ジョブ制

御に関連するシグナルには“job”を記してある。

デフォルト動作が「coreを作成して終了」の場合、プロセスのカレント作業ディレクトリにプロセスのメモリイメージをcoreというファイル名で残すことを意味する。(coreというファイル名から、UNIXにはこの機能が長い間存在していることが分かる。) UNIXのほとんどのデバッガでは、このファイルからプロセスの終了時の状態を調べられる。なお、(a) セットユーザIDされたプロセスであり、しかも、カレントユーザとプログラムファイルの所有者が異なる場合、(b) セットグループIDされたプロセスであり、しかも、カレントユーザとファイルのグループが異なる場合、(c) カレント作業ディレクトリに書き込み許可を持たないユーザの場合、(d) ファイルが大きくなりすぎる (7.11節のRLIMIT_COREリミット) 場合、のいずれかの場合には、coreファイルを作らない。coreファイルのパーミッションは (既存でないと仮定して)、通常、ユーザは読み書き、グループは読み取りのみ、その他も読み取りのみである。

> coreファイルの生成は、UNIXのほとんどのバージョンの実装上の機能である。POSIX.1には含まれていない。
>
> UNIXバージョン6では、条件 (a) と (b) を検査せず、ソースコードにはつぎのような注釈が付いていた。「プロテクションの抜け穴を探しているなら、セットユーザIDコマンドがcoreを生成すると見つかるであろう。」
>
> 4.3+BSDでは、実行したプログラム名のはじめの16文字を *prog* として、core.*prog* という名前のファイルを生成する。coreファイルの識別に便利な機能である。

図10.1に「ハードウェアフォルト」と書いたシグナルは、実装に依存したハードウェアフォルトに対応する。これらの名前のほとんどは、UNIXをはじめて実装したPDP-11からとられている。これらのシグナルがどのような種類のエラーに対応するかは、読者のシステムのマニュアルで確認してほしい。

以下に各シグナルを詳しく説明する。

SIGABRT　このシグナルはabort関数 (10.17節参照) を呼ぶと生成される。プロセスは異常終了する。

SIGALRM　このシグナルは、alarm関数で設定したタイマーが切れると生成される。詳しくは、10.10節を参照。

setitimer(2) 関数で設定したインターバルタイマーが切れた場合にも、このシグナルが生成される。

SIGBUS　実装で定義されたハードウェアフォルトを表す。

SIGCHLD　プロセスが終了したり休止したりすると、親に対してSIGCHLDシグナルが送られる。デフォルトではこのシグナルを無視するため、親側で子の状態変化を知りたいときには、このシグナルを捕捉する必要がある。シグナル捕捉関数での一般的な動作は、wait関数の1つを呼んで子のプロセスIDと終了状態を取得することである。

システムVの初期のリリースには、類似した名前のシグナルSIGCLD (Hが…がある。このシグナルには非標準の意味があり、SVR2にまで遡っても、マニ…ページには新規プログラムはこれを使わないように警告している。アプリケー…では、標準のSIGCHLDシグナルを使うべきである。10.7節でこの2つのシグ…説明する。

SIGCONT　ジョブ制御に関連したこのシグナルは、休止プロセスの実行を続行させたいときにプロセスに送る。プロセスが休止している場合、デフォルト動作はプロセスを続行することであり、それ以外の場合、デフォルト動作はシグナルを無視することである。例えば、viエディタはこのシグナルを捕捉して端末画面を再描画する。詳しくは10.20節を参照してほしい。

SIGEMT　実装で定義されたハードウェアフォルトを表す。

EMTという名前は、PDP-11の「エミュレータトラップ (emulator trap)」命令に由来する。

SIGFPE　このシグナルは、0除算、浮動小数点のオーバーフローなどの算術演算例外である。

SIGHUP　端末インタフェースで接続断を検出した場合に、このシグナルは制御端末に結び付けられた制御プロセス (セッションリーダ) に送られる。図9.11のsession構造体のs_leaderメンバーが指すプロセスに対してこのシグナルが送られる。端末のCLOCALフラグが設定されていない場合にかぎり、条件が揃うとこのシグナルが生成される。(接続された端末がローカルの場合に、端末のCLOCALフラグが設定される。これは、端末ドライバにモデムの状態信号をすべて無視させる。第11章でこのフラグの設定方法を述べる。) このシグナルを受け取るセッションリーダは、バックグラウンドである場合もある。例えば、図9.7を参照してほしい。常にフォアグラウンドプロセスグループに対して送られる端末生成のシグナル (割り込み、クイット、サスペンド) とは異なる。

セッションリーダが終了した場合にも、このシグナルが生成される。この場合、フォアグラウンドプロセスグループの各プロセスに対してこのシグナルが送られる。

このシグナルは、デーモンプロセス (第13章) に構成設定ファイルを読み直させるためにしばしば用いられる。SIGHUPが選ばれた理由は、デーモンには制御端末がないためこのシグナルを決して受け取ることはないからである。

…IGILL
このシグナルは、プロセスが不正なハードウェア命令を実行したことを表す。

4.3BSDでは、abort関数はこのシグナルを生成する。現在はSIGABRTを使っている。

…IGINFO
この4.3+BSDのシグナルは、端末の状態表示キー (しばしばControl-T) を押すと端末ドライバが生成する。このシグナルは、フォアグラウンドプロセスグループのすべてのプロセスに対して送られる (図9.8を参照)。このシグナルは、フォアグラウンドプロセスグループのプロセスに対して状態の情報を端末に表示させる。

…INT
端末の割り込みキー (しばしば、DELETEやControl-C) を押すと、端末ドライバが生成するシグナルである。このシグナルは、フォアグラウンドプロセスグループのすべてのプロセスに対して送られる (図9.8を参照)。このシグナルは暴走しているプログラムを終了するためにしばしば使われ、不必要な大量の出力を画面に表示しているような場合には特に有用である。

このシグナルは非同期入出力事象を表す。これについては、12.6.2節で説明する。

図 10.1 には、SIGIO のデフォルト動作を終了または無視と記した。残念ながらデフォルト動作はシステムに依存する。SVR4 では、SIGIO は SIGPOLL と等価であり、デフォルト動作はプロセスを終了する。4.3+BSD (このシグナルは 4.2BSD で導入された) では、デフォルト動作は無視である。

SIGIOT　実装で定義されたハードウェアフォルトを表す。

IOT という名前は、PDP-11 のニーモニック (mnemonic)「入出力トラップ (input/output TRAP)」命令に由来する。

システム V の初期のバージョンでは、abort 関数はこのシグナルを生成していた。現在は SIGABRT を使っている。

SIGKILL　このシグナルは、捕捉したり無視できない 2 つのシグナルのうちの 1 つである。システム管理者に任意のプロセスを終了させる確実な手段を提供する。

SIGPIPE　読み手が終了しているパイプラインに書き出すと、SIGPIPE が生成される。パイプについては 14.2 節で説明する。反対側の受け手が終了してしまったソケットに書いた場合にも、このシグナルが生成される。

SIGPOLL　この SVR4 のシグナルは、ポーリング可能な装置において特定の事象が発生すると生成される。このシグナルについて、12.5.2 節の poll 関数で述べる。おおまかには、4.3+BSD の SIGIO や SIGURG シグナルに対応する。

SIGPROF　setitimer(2) 関数で設定したプロフィールインターバルタイマーが切れるとこのシグナルが生成される。

SIGPWR　この SVR4 のシグナルはシステムに依存する。無停電電源 (UPS) を装備したシステムで主に用いられる。給電が停止すると無停電電源がとって代わり、このことは通常ソフトウェアに通知される。この時点では、システムはバッテリ電源で稼働し続けるため、何も処置する必要はない。しかし (給電が長い期間停止しているなどして) バッテリの能力が低下すると、ソフトウェアは再度通知を受ける。この時点で、システムは 15 から 30 秒程度の間にすべてをシャットダウンする義務を負う。このときに SIGPWR が送られる。ほとんどのシステムには、バッテリの機能低下の通知を受けるプロセスがあり、このプロセスが init プロセスに SIGPWR シグナルを送り、init がシャットダウン処理を行う。システム V の init の多くの実装では、この目的のために powerfail と powerwait の 2 つの項目が inittab ファイルのなかにある。

RS-232 によるシリアル接続で、コンピュータにバッテリ能力の低下を通知できる低価格の無停電電源が利用できるようになったため、このシグナルがより重要になりつつある。

SIGQUIT　端末のクイットキー (しばしば Control-\) を打つと端末ドライバが生成するシグナルである。フォアグラウンドプロセスグループのすべてのプロセスに対してこのシグナルが送られる (図 9.8 を参照)。このシグナルは、(SIGINT のように) フォアグラウンドプロセスグループを終了させるだけでなく、core ファイルも生成する。

SIGSEGV　プロセスが不正なメモリ参照を行ったことを表すシグナルである。

名称 SEGV は、"SEGmentation Violation"の略である。

SIGSTOP　ジョブ制御に関連したこのシグナルは、プロセスを休止させる。これは対話的な休止シグナル (SIGTSTP) に似ているが、SIGSTOP は捕捉も無視もできない。

SIGSYS　不正なシステムコールを知らせる。カーネルがシステムコールと解釈するような機械語命令をプロセスは実行したが、システムコールの種類を表すパラメータが不正であった場合など。

SIGTERM　kill(1) コマンドがデフォルトで送る終了シグナルである。

SIGTRAP　実装で定義されたハードウェアフォルトを表す。

シグナル名称は、PDP-11 のトラップ命令に由来する。

SIGTSTP　端末のサスペンドキー (しばしば Control-Z) を押すと、端末ドライバが生成する対話的な休止シグナルである†。

フォアグラウンドプロセスグループのすべてのプロセスに対して、このシグナルは送られる (図 9.8 を参照)。

SIGTTIN　バックグラウンドプロセスグループのプロセスが制御端末から入力しようとすると、端末ドライバがこのシグナルを生成する。(これについては 9.8 節の説明を参照。) 特別な場合として、(a) 入力しようとしているプロセスがこのシグナルを無視したりブロックしている場合、(b) 入力しようとしているプロセスのプロセスグループがオーファンドの場合、このシグナルは生成されない。その代わり、入力操作は errno に EIO を設定してエラーを戻す。

SIGTTOU　バックグラウンドプロセスグループのプロセスが制御端末へ出力しようとすると、端末ドライバがこのシグナルを生成する。(これについては 9.8 節の説明を参照。) 上述した SIGTTIN シグナルと異なり、バックグラウンドが制御端末へ出力可能かどうかはプロセスが選択できる。この選択の変更方法は第 11 章で述べる。

バックグラウンドの出力が許されていない場合、SIGTTIN シグナルと同じように特別な場合が 2 つある。つまり、(a) 出力しようとしているプロセスがこのシグナルを無視したりブロックしている場合、(b) 出力しようとしているプロセスのプロセスグループがオーファンドの場合、このシグナルは生成されない。その代わり、出力操作は errno に EIO を設定してエラーを戻す。

バックグラウンドの出力の許可のあるなしにかかわらず、(出力以外の) 特定の端末操作を行っても SIGTTOU シグナルが生成される。tcsetattr、tcsendbreak、tcdrain、tcflush、tcflow、tcsetpgrp である。これらの端末操作については第 11 章で述べる。

URG　緊急状態が発生したことをプロセスに通知するシグナルである。ネットワーク接続を介して緊急データが到着した場合にもオプションで生成される。

ER1　アプリケーションプログラムが使用するユーザ定義のシグナルである。

† 休止 (stop) という用語は混乱を招く場合がある。ジョブ制御とシグナルについて説明している場合、これはジョブの休止と続行に関したことである。一方、端末ドライバの場合、休止という用語は歴史的に端末への出力を Control-S と Control-Q の文字を用いて休止したり再開することを意味する。このため、対話的な休止シグナルを生成する文字を、端末ドライバではサスペンド文字と呼び、休止文字とは呼ばない。

SIGUSR2　アプリケーションプログラムが使用するユーザ定義のシグナルである。

SIGVTALRM　setitimer(2) 関数で設定した仮想インターバルタイマーが切れた場合に生成されるシグナルである。

SIGWINCH　SVR4 と 4.3+BSD のカーネルは、各端末と擬似端末に結び付けられたウィンドウサイズを管理している。11.12 節で述べる ioctl 関数を用いて、プロセスはウィンドウサイズを取得したり設定できる。プロセスが ioctl のウィンドウサイズ設定コマンドでウィンドウサイズを変更すると、カーネルはフォアグラウンドプロセスグループに対して SIGWINCH シグナルを生成する。

SIGXCPU　SVR4 と 4.3+BSD ではリソースリミットの概念を使える (7.11 節を参照)。プロセスがソフト CPU 時間リミットを超過すると、SIGXCPU シグナルが生成される。

SIGXFSZ　プロセスがファイルサイズのソフトリミットを超過すると、SVR4 と 4.3+BSD ではこのシグナルが生成される (7.11 節を参照)。

10.3 signal 関数

UNIX のシグナル機能に対するもっとも簡単なインタフェースは signal 関数である。

```
#include <signal.h>

void (*signal(int signo, void (*func)(int)))(int);
```

戻り値: 前に設定されたシグナル処理 (以下を参照)

signal 関数は ANSI C で定義されている。ANSI C には、複数プロセス、プロセスグループ、端末入出力などの概念がないため、ANSI C のシグナルの定義はほとんどの UNIX システムに対しては無用なほど曖昧である。ANSI C におけるシグナルの記述はたった 1 ページであるが、POSIX.1 では 15 ページを割いている。

SVR4 にも signal 関数があるが、SVR4 でこれを用いると SVR2 の信頼性のない古い意味のシグナルを使ったことになる。(これらの古い意味については 10.4 節で述べる。古い意味を必要とするアプリケーションとの互換性を保つためにこれらの関数がある。新しいアプリケーションでは、これらの信頼性のないシグナルを使用すべきではない。

4.3+BSD にも signal 関数があるが、これは (10.14 節で述べる) sigaction 関数を用いて定義されている。したがって、4.3+BSD で signal 関数を用いた場合には、信頼性のある古い意味のシグナルを使うことになる。

sigaction 関数を説明するときに、signal の実現方法を述べる。本書のすべてのプログラムは、プログラム 10.12 に示した signal 関数を用いる。

signo 引数には、図 10.1 のシグナル名を指定する。*func* の値は、(a) 定数 SIG_IGN、(b) 定数 SIG_DFL、(c) シグナル発生時に呼び出す関数のアドレス、である。SIG_IGN は、シグナルを無視するようにシステムに指示する。(2 つのシグナル、SIGKILL と SIGSTOP は無視できないことに注意。いつも捕捉して終了させてほしい。) SIG_DFL は、シグナルの動作としてデフォルト動作を行う指示である (図 10.1 の最後の欄を参照)。シグナル発生時に呼び出すべき関数のアドレスを指定することを、シグ

「捕捉 (catch)」するという。この関数を、シグナルハンドラ (signal handler) とか、シグナル捕捉関数 (signal-catching function) と呼ぶ。

signal 関数のプロトタイプから、この関数には 2 つの引数が必要であり、戻り値を返さない (つまり、void を返す) 関数へのポインタを返すことが分かる。はじめの引数 *signo* は整数である。第 2 引数は、整数引数を 1 つ取り戻り値を返さない関数を指すポインタである。signal の戻り値として返されるアドレスが指す関数は、整数引数を 1 つ取る (最後の (int))。この宣言を簡単に言い換えると、シグナルハンドラには整数引数が 1 つ (シグナル番号) 渡され、戻り値を返さないということである。シグナルハンドラを設定するために signal を呼ぶときの第 2 引数は、関数を指すポインタである。signal の戻り値は、それ以前に設定されていたシグナルハンドラを指すポインタである。

> 多くのシステムでは、シグナルハンドラを呼び出す場合、実装に依存した引数が追加される。SVR4 と 4.3+BSD におけるオプションの引数については、10.21 節で述べる。

本節のはじめに示した signal 関数の複雑なプロトタイプは、つぎの typedef[Plauger 1992] を用いると簡潔になる。

```
typedef void    Sigfunc(int);
```

とすれば、

```
Sigfunc *signal(int, Sigfunc *);
```

となる。この typedef は ourhdr.h (付録 B 参照) に収めてあり、本章の関数で用いている。

システムのヘッダー <signal.h> を調べると、つぎのような形式の宣言がある。

```
#define SIG_ERR   (void (*)())-1
#define SIG_DFL   (void (*)())0
#define SIG_IGN   (void (*)())1
```

これらの定数は、signal の第 2 引数に指定する「整数引数を 1 つ取り値を返さない関数を指すポインタ」や signal の戻り値に使用できる。これらの定数に使われる 3 つの値は、−1、0、1 である必要はない。関数を指すアドレス以外の値であればよいのである。ほとんどの UNIX システムでは上に示した値を用いている。

プログラム例

プログラム 10.1 は、2 つのユーザ定義シグナルを捕捉してシグナル番号を出力する簡単なシグナルハンドラである。呼び出したプロセスをスリープさせる pause 関数については、10.10 節で述

プログラム 10.1　SIGUSR1 と SIGUSR2 を捕捉する簡単なプログラム

```
lude    <signal.h>
lude    "ourhdr.h"

void sig_usr(int);    /* one handler for both signals */
```

```
main(void)
{
    if (signal(SIGUSR1, sig_usr) == SIG_ERR)
        err_sys("can't catch SIGUSR1");
    if (signal(SIGUSR2, sig_usr) == SIG_ERR)
        err_sys("can't catch SIGUSR2");

    for ( ; ; )
        pause();
}

static void
sig_usr(int signo)      /* argument is signal number */
{
    if (signo == SIGUSR1)
        printf("received SIGUSR1\n");
    else if (signo == SIGUSR2)
        printf("received SIGUSR2\n");
    else
        err_dump("received signal %d\n", signo);
    return;
}
```

プログラムをバックグラウンドで起動し、kill(1) コマンドを用いてシグナルを送る。UNIX の *kill* という用語は呼び誤りである。kill(1) コマンドと kill(2) 関数は、プロセスやプロセスグループにシグナルを送るだけである。そのシグナルがプロセスを終了させるかどうかは、送られたシグナルとプロセスがそのシグナルを捕捉するかどうかに依存する。

```
$ a.out &                       プロセスをバックグラウンドで始動する
[1]       4720          ジョブ制御を行うシェルはジョブ番号とプロセス ID を出力する
$ kill -USR1 4720               SIGUSR1 を送る
received SIGUSR1
$ kill -USR2 4720               SIGUSR2 を送る
received SIGUSR2
$ kill 4720                     SIGTERM を送る
[1] + Terminated        a.out &
```

SIGTERM シグナルを送ると、プロセスはこのシグナルを捕捉せず、シグナルのデフォルト動作が了であるためプロセスは終了する。□

●プログラムの始動●

プログラムが exec されたとき、すべてのシグナルに対する動作は、デフォルトのものを無視するかである。exec を呼び出したプロセスがシグナルを無視していないかぎり、すべてグナルにはデフォルト動作が設定される。つまり、exec 関数は、捕捉するシグナルに対してフォルト動作を設定し、それ以外のシグナルについては設定を変更しない。(exec を呼び出ロセスで捕捉していたシグナルを、新しいプログラム側で捕捉することは普通できない。exec を呼び出した側のシグナル捕捉関数のアドレスは、exec された新しいプログラムでは無意味だからである。)

(ほとんど認識していないであろうが) 日常的に出合う例は、バックグラウンドプロセスに対する割り込みやクイットシグナルを、対話的シェルがどのように扱っているかである。ジョブ制御を行わないシェルでは、つぎのようにプロセスをバックグラウンドで実行すると、バックグラウンドプロセスでは割り込みやクイットシグナルを無視するようにシェルが自動的に設定する。

```
cc main.c &
```

これは、割り込み文字をタイプしても、バックグラウンドジョブになんら影響しないようにするためである。これを行わないと、割り込み文字をタイプしたときに、フォアグラウンドプロセスを終了するだけでなく、すべてのバックグラウンドプロセスをも終了してしまう。

これら 2 つのシグナルを捕捉する対話的なプログラムの多くには、つぎのようなコードがある。

```
int   sig_int(), sig_quit();

if (signal(SIGINT, SIG_IGN) != SIG_IGN)
    signal(SIGINT, sig_int);
if (signal(SIGQUIT, SIG_IGN) != SIG_IGN)
    signal(SIGQUIT, sig_quit);
```

これにより、プロセスは現在無視していないシグナルのみを捕捉する。

これら 2 つの signal の呼び出しは、signal 関数の制限を示すものでもある。つまり、シグナルに対する動作を変更せずに、シグナルに対する現在の動作を判定できないのである。本章では、設定を変更せずにシグナルに対する動作を調べることができる sigaction 関数について後ほど説明する。

●プロセスの生成●

プロセスが fork を呼ぶと、子は親のシグナル動作を継承する。子は親のメモリイメージを複製して動作し始めるため、子においてもシグナル捕捉関数のアドレスは意味を持

10.4 信頼性のないシグナル

UNIX の (バージョン 7 のような) 初期のバージョンでは、シグナルには信頼性がなかった。シグは生起するがプロセスには伝えられない場合がある。つまり、シグナルを紛失する場合があっである。また、プロセスはシグナルをほとんど制御できなかった。つまり、シグナルを捕捉す視するだけである。シグナルをブロックするようにカーネルに指示したい場合がある。つまグナルを無視するのではなく、シグナルが生起したことを記録しておき、準備が整った後にてほしいのである。

4.2BSD で信頼性のあるシグナルに変更された。システム V においては、信頼性のあるシグナルを提供するための別の種類の変更が SVR3 で行われた。POSIX.1 は、BSD のモデルを規格として採用している。

バージョンにおける 1 つの問題点は、シグナルが発生する度にシグナルに対する動作がデ動作に戻ることである。(プログラム 10.1 を実行した前節では、シグナルを 1 回だけ捕捉このような詳細に立ち入ることを避けた。) このような初期のシステムを対象とした

CHAPTER

14

プロセス間通信

14.1 はじめに

第8章では、プロセス制御の基本操作について述べ、複数プロセスを起動する方法を見た。しかし、これらのプロセス間で情報を交換する手段は、オープンしておいたファイルを fork や exec で渡すか、ファイルシステムを介すのみであった。ここでは、プロセスが互いに通信するための別の方法について述べる。IPC、つまり、プロセス間通信 (interprocess communication) である。UNIX の IPC は、異なる方式の寄せ集めであり、UNIX のすべての実装においてポータブルなものは存在しない。図 14.1 は、異なる実装において使用できる異なる方式の IPC をまとめたものである。

図 14.1 UNIX の IPC のまとめ

IPC の方式	POSIX.1	XPG3	V7	SVR2	SVR3.2	SVR4	4.3BSD	4.3+BSD
(半二重) 前付きパイプ)	• •	• •	•	• •	• •	• •	•	• •
ムパイプ (全二重) ストリームパイプ					• •	• •	• •	• •
キュー		• • •		• • •	• • •	• • •		
					•	• •	•	•

かるように、UNIX の実装にかかわらず信頼して使用できる唯一の IPC は半二重のパ