情報爆発時代の耐障害性OS支援の一考察

島田 裕正† 石川 広男‡ 中島 達夫†

概 要

アプリケーションを構成する一部のプロセスが障害によって停止した場合、アプリケーションはプロセスを始動することによって復旧できる。しかし、その方法はアプリケーションに依存している。本稿ではLinux上で自動再起動可能な性質を持つプロセスを提案する。これにより、障害復旧処理をアプリケーションから非依存にすることが可能になる。自動再起動プロセスに関する一連の仕組みをユーザレベルおよびカーネルレベルでそれぞれ実装する。評価では、二つの実装を比較する。

1 序論

近年、オペレーティングシステムレベルでの信頼性支援が、個人のシステムにおいても注目されつつある[1]。これは組み込み機器に今までより高機能で複雑なオペレーティングシステムが搭載されるようになってきたために、障害が顕在化したためである。障害に対する一時的な対処方法として、システムの再起動が有効であると言われているが、システムが再び利用可能になるまでの時間が長いことが問題である[2]。

アプリケーションにおける対策は、アプリケーションを複数プロセスに分割して障害を隔離し、プロセス単位での再起動を可能にする方法である。システム全体の再起動を避けることが可能であることに加えて、ファイルへのアクセス権を分割したプロセス毎に設定できることから[3][4]、安全性の観点からも望ましい構成である。しかし、各プロセスをアプリケーション自身が監視しなければならないため、アプリケーションが複雑になることが問題である。

本稿はアプリケーションから独立したプロセス再始動機構Rebootable fork(rfork)を提案する。rforkは自動的に再起動するプロセスを生成するAPIである。rforkを使うことでアプリケーションは再起動の仕組みをそれ自身で持つ必要がなくなる。関連研究[2]においても、アプリケーションを構成するコンポーネント単位での再起動を提案しているが、ミドルウェア上の実装であり、ファイルへのアクセス権などをOSで制御できない点が我々の提案と異なる。

次章では、rforkの設計と実装について述べる。第3章では二つの実装の評価と比較を行う。第4章で結論を述べる。

†早稲田大学基幹理工学部情報理工学科
‡早稲田大学IT研究機構

2 Rebootable forkの設計と実装

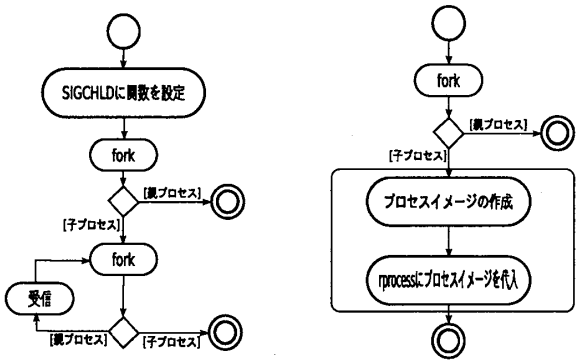
rforkは自動再起動可能なプロセスを生成する。forkで生成されたプロセスは、親プロセスにシグナルを送り終了する。しかし、rforkで生成されたプロセスは、親プロセスにシグナルを送り、再起動する。再起動時には、初回と同じ引数が再びプロセスに渡される。

rforkはその終了状態によってプロセスを再起動させるかどうか決定する。プロセスが自発的に再起動を要求する場合とカーネルがプロセスのエラーを検知した場合には、プロセスは再起動される。プロセスが正常終了した場合には、そのプロセスは削除される。

本稿ではユーザレベルとカーネルレベルでこのrforkを実装する。ユーザレベルではrforkをライブラリとして実装し、カーネルレベルではシステムコールとしてrforkを実装する。カーネルレベルで実装する際に、task_struct構造体にrprocessという変数を追加した。rprocessはtask_struct構造体のポインタで再起動する際にこの変数が指してあるプロセスイメージを起動することで子プロセスは再起動する。

2.1 ユーザレベルの実装

ユーザレベルrforkは内部で二度、forkシステムコールを呼び出し、子と孫のプロセスを生成する（図1(a)）。子プロセスは孫プロセスを監視し、孫プロセスの再起動を行う。孫プロセスがユーザプログラムを実行する。孫プロセスの障害を検知するため、子プロセスはSIGCHILDシグナルを受け取るように設定される。孫プロセスの終了時の値も子プロセスで検査され、再起動の有無を判断する。



(a) ユーザレベル　(b) カーネルレベル

図 1: rfork の処理の流れ

OS	Linux Debian lenny
カーネル	linux-kernel-2.6.23
CPU	Intel Core2 Quad
メモリ	4096MB

表 1: 計測環境

	ユーザ	カーネル
平均値 [マイクロ秒]	310.4	295.63
標準偏差	64.84	20.63

表 2: 計測結果

2.2 カーネルレベルの実装

カーネルレベル rfork はプロセスイメージを複製し、それを使ってプロセスの再起動を実現する（図 1(b)）。複製を保持するために task_struct 構造体を拡張した。再起動の判断は、do_exit 関数の中で行うように拡張した。

3 評価

始めに使用制限について二つの rfork を比較する。ユーザレベル rfork ではプログラムの SIGCHLD に対して設定をすることができない。これは rfork で SIGCHLD の設定をしているからである。一方カーネルレベル rfork ではカーネルに rfork システムコールを埋め込まなければならない。従って、ユーザレベル rfork の方が移植性が高いが、カーネルレベル rfork のほうが柔軟なプログラムを書くことが出来る。

次にオーバヘッドについて比較を行う。ユーザレベル rfork では再起動の情報を受信するためにプロセスを余分に一つ作っている。一方カーネルレベル rfork では、子プロセスを起動する際に再起動時に起動させるプロセスイメージを一つ作成する。二つとも子プロセスが起動しているときに保持している余分なプロセスイメージの数は同じである。しかし、カーネルレベル rfork の作成したプロセスイメージは再起動が必要になったとき初めてスケジューリングに組み込まれるので、ユーザレベルの rfork と比べるとオーバヘッドは少ないと考えられる。

本稿では簡単なプログラムを使って性能の計測を行った。この計測プログラムは rfork を呼び出してから現在の時刻を表示し、再起動するように exit 関数の引数に指定した値を与えて再起動させるものである。今回は 1000 回再起動させ、それにかかる時間を集計した。本稿で計測に用いた環境は表 1 の通りである。計測結果は表 2 の通りである。

4 結論

今回ユーザレベルとカーネルレベルの二つの実装を行った。ユーザレベルで作成した rfork は移植性が高く、POSIX 環境の OS であればこのライブラリを使うことが出来る。一方カーネルレベルで作成した fork は OS が自動再起動プロセスを生成する機能を提供することになる。計測結果からカーネルレベル rfork の方が安定していてなおかつ速度も早いことが分かる。したがって、性能はカーネルレベル rfork の方がよいといえる。

参考文献

[1] Feng Qin, Joseph Tucek, Yuanyuan Zhou, and Jagadeesan Sundaresan. Rx: Treating Bugs As Allergies–A Safe Method to Survive Software Failures. *ACM Transactions on Computer Systems*, vol.25, no.3, August 2007.

[2] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot – A Technique for Cheap Recovery. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, December 2004.

[3] Maxwell Krohn. Building Secure High-performance Web Services with OKWS. In *Proceedings of the 2004 USENIX Annual Technical Conference (USENIX '04)*, June 2004.

[4] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing Privilege Escalation. In *Proceedings of the 12th USENIX Security Symposium*, August 2003.