실시간 내장 시스템 환경에서의 빠른 프로세스 복구 기법

김광식, 유준석†, 류준길, 박찬익
시스템 소프트웨어 연구실, 시스템 & 네트워크 연구실† 포항공대
{honesty,jsyoo, lancer, cipark}@postech.ac.kr

Fast Process Recovery Technique In Real-Time Embedded System

Kwangsik Kim, Junseok Yoo†, Junkil Ryu, Chanik Park
System Software Lab., System & Network Lab.† POSTECH

요 약 내장 시스템(Embedded System)기술은, 정부가 주도하는 기술과제로 여러 응용분야에서 각광을 받고 있다. 본 논문에서는 내장 시스템이 가지는 한계상황 하에서 프로세스가 좀더 빠르게 복구하는 기법을 제안하고자 한다. 빠른 복구를 위해서는 두 가지 조건이 만족되어야 한다. 첫째 조건은 실제 프로세스의 이상이 발생 했는지를 빠르게 감지해야 한다. 기존에는 주기적으로 프로세스를 감시하는 방법들이 많이 사용되었으나 이런 방법들은 내장 시스템에서 빠른 프로세스 복구를 하는데 한계점들이 나타났다. 따라서 시스템 레벨에서 프로세스 종료를 시키는 시그널(signal)을 훔치는(hooking) 방법[1]과 프로세스 스케줄 순서를 조정하는 방법을 토대로 프로세스의 이상을 빠르게 감지할 수 있다. 두 번째는 한정된 자원 아래서 효율적으로 복구 데이터를 관리 및 복구해야 한다. 기존의 복구 기법에 경우 다양한 자원을 대한 복구를 위해서 자원을 많이 사용하였지만 우리가 사용하는 공유메모리 기법[1]은 자신의 필요한 정보만을 관리함으로써 한정된 자원 환경에서 복구가 가능하도록 하였다.

1. 서 론

최근에 내장 시스템(Embedded System)기술에 대한 관심이 늘어가고 있다. 이러한 관심은 내장 시스템 기기에게 다양한 기능들을 요구한다. 이로 인해서 내장 시스템에 운영 시스템의 도입은 선택 조건이 아닌 필수 조건이 되고 있다. 그러나 운영 시스템의 도입은 시스템의 멀티태스킹 기능은 향상시켜줬지만 시스템의 실시간성에는 역기능을 가져왔다. 내장 시스템에서 작동하는 프로세스들이 실시간성이 부족해서 제대로 원하는 처리를 하지 못하는 경우가 발생하곤 한다. 그러므로 운영 시스템에서의 실시간성 향상은 대부분의 내장 시스템의 운영 시스템 도입의 중요한 역할을 할 것이다. 또한 이러한 실시간성은 프로세스가 오류가 나서 복구를 할 경우에도 필요하다.

우리는 크래쉬가 발생하였을 때 기존의 시그널을 통해 빠른 에러 감지와 그에 따르는 복구 기법을 개선하여, 실시간성을 요구하는 프로세스 환경에서 빠른 복구 기술을 개발하였다. 그리고 이런 복구 기술 구현을 통해서 전체적인 시스템의 실시간성이 향상된 것을 확인할 수 있었다. 2장에서는 이런 분야에 관련된 연구에 대해서 간단히 살펴보고 3장에서는 제안된 시스템 구성에 대한 설명을 하게 될 것이다. 4장과 5장에서는 구현과 실험에 대해서 설명할 것이다. 6장에서 결론을 맺을 것이다.

2. 관련 연구

프로세스에 크래쉬가 발생하였을 때 프로세스를 정상적으로 복구하기 위해서는 크래쉬의 발생을 감지하는 부분과 프로세스를 복구하는 부분으로 나누어 질 수 있다.

프로세스의 크래쉬를 감지하는 기술은 크게 두 가지로 나눌 수 있다. 데몬 형식으로 크래쉬를 감지하는 방법과 시그널을 통해서 크래쉬를 감지하는 방법이 있다.

데몬 형식으로 크래쉬를 감지하는 방법은 비실시간성 응용프로그램에서 널리 사용되는 기술로 이 기술은 run-time overhead와 크래쉬 감시 시간이 크기 때문에 실시간성을 요구하는 내장 시스템에서 이용하기에는 무리가 있다. 그리고 크래쉬 감시 시간을 줄이기 위해 모든 제어 루프상의 프로세스를 체크하는 방법을 사용하는데, 충분히 자주 체크되지 않을 경우에 체크가 누락된 크래쉬는 복구를 지연시키고 결국 deadline miss를 많이 일으키게 된다.

이러한 문제점을 극복하기 위해서 실시간성을 요구하는 내장 시스템에서는 크래쉬 통지 시그널(Fault notification signal)을 사용하여 크래쉬를 감지하게 된다. 복구를 요구하는 모든 크래쉬 관련 시그널들을 위한 특별한 시그널 핸들러를 할당하여 단순하게 시그널 핸들러만 수행하여 복구를 수행할 수 있다.[1] 이런 방식에서 복구를 필요로 하는 프로세스와 다른 프로세스들이 같이 동작 시에는 복구를 필요로 하는 프로세스에 크래쉬 관련 시그널이 생성된 후에도 해당 시그널을 처리하기까지는 오랜 시간이 걸릴 수 있다. 따라서 제안된 기법에서는 크래쉬 관련 시그널 핸들러가 즉각적으로 수행이 될 수 있도록 스케줄 순서를 조정함으로써 빠른 감지를 하게 되었다.

다음으로 시그널 핸들러에서 수행하게 되는 프로세스 복구 단계에서는 실시간성을 요구하는 내장 시스템에서 저장공간 제약성을 만족시켜야 한다. 이와 관련된 복구 방법으로는 backward error recovery 기법으로 알려진 체크 포인팅(check pointing)과 롤백(rollback)을 이용하는 방법[5,6,7] 등이 있다. 하지만 이런 기법들은 내장 시스템이 가지는 저장공간 제약성을 만족시키기 어렵다. 따라서 공유메모리를 이용해서 필요한 정보만을 저장하는 기법을 사용하였다[1].

3. 제안된 프로세스 복구 시스템

- 크래쉬가 발생한 프로세스 감지 기법

제안된 크래쉬 감지 기법은 응용 프로그램에서는 주기적으로 프로세스의 상태 혹은 정보 교환을 통해서 탐지하는 방식이 아니다. 주기적으로 탐지하는 경우에는 프로세스의 문제를 찾고자 할 때 평균적으로 감지 시간이 한 프로세스의 실행

주기의 두 배 이상이 된다. 리눅스 커널 버전 2.4일 경우에는 한 프로세스의 실행 주기의 경우에는 100ms이다. 따라서 200ms이상의 시간이 걸려야 실제로 크래쉬 탐지 후에 이에 대한 대응이 가능하게 된다. 즉 빠른 감지가 힘들게 된다.

우리는 시그널을 통해서 크래쉬를 감지하는 방식을 사용한다. 유닉스 계열의 시스템들은 프로세스가 크래쉬를 발생시킬 경우, 해당 프로세스에게 어떤 종류의 에러가 발생했는지를 시그널을 통해서 알려준다. 우리는 시스템 차원에서 시그널이 발생하는 것을 감지함으로써 해당 프로세스에 크래쉬가 발생했는지를 알 수 있게 된다. 하지만 시그널이 발생했다고 해서 모든 시그널들이 다 크래쉬를 발생하는 시그널은 아니다. 시그널들 중에는 작업 제어와 관련된 정상적인 시그널들도 존재한다. (예, SIGINT, SIGTSTP, SIGKILL, SIGCONT, 등.) 하지만 우리는 이런 시그널 정보를 엿봄(hooking)으로써 시스템에서 우리가 복구를 원하는 프로세스에 크래쉬와 관련된 시그널들을 발생하는 지를 감시할 수 있게 된다.

POSIX에서는 다양한 시그널에 대해서 정의를 해놓았다. 그 중에는 크래쉬와 관련된 시그널만 아래의 표 1. 에서 보여주고 있다.

시그널	세부 설명
SIGFPE	Erroneous arithmetic operation.
SIGILL	Illegal instruction.
SIGSEGV	Invalid memory reference.
SIGBUS	Access to an undefined portion of a memory object.
SIGSYS	Bad system call.
SIGXCPU	CPU time limit exceeded.
SIGXFSZ	File size limit exceeded.

표 1 프로세스가 크래쉬일 때 알려주는 시그널들

- 크래쉬가 발생한 프로세스 복구 기법

아래 표에서는 각각의 복구 시스템들의 지원 정도를 보여 주고 있다. 또 표 2 에서 알 수 있듯이 복구 시스템들은 다양한 복구 기능을 제공해 주고 있다.

Name	*Type*	*Scope*	*File Data*	*Resource Usage*	*Credentials*	*Checkpoint Handlers*	*Signals*	*File Descriptors*	*Address Space*	*Registers*
libckp	lib	Process	-	-	-	-	-	○	○	●
libckpt	lib	Process	○	-	-	-	-	○	○	●
Condor	lib	Process	-	-	-	△	●	○	○	●
libckpt	lib	Thread	-	-	-	○	●	○	○	●
CRAK	sys	Child	-	-	○	△	●	○	○	●
BPRoc	sys	Process	-	○	△	-	●	-	○	●
Score	lib	Parallel	-	-	-	○	●	○	○	●
CoCheck	lib	Parallel	-	-	-	△	●	○	○	●
- = Missing. △ = Weak. ○ = Good. ● = Complete										

표 2 복구 시스템의 성능 비교

Libckpt[8]에서는 복귀를 위한 이미지 파일을 생성하게 된다. 실제 이 파일의 경우에는 우리가 원하는 복구 파일의 바이너리 사이즈보다 훨씬 큰 이미지 정보를 가지게 된다. 그 이유는 복구 시스템의 경우에는 실제 프로세스가 어떤 정보를 필요한지에 대한 아무런 정보도 가지고 있지 않으므로 표 2 에서 나오는 모든 정보를 저장 해야 한다. 그런데 내장 시스템은 작은 사이즈의 메모리를 가지고 있기 때문에 복구를 위해서 대용량의 정보를 저장할 수 없다. 따라서 앞에서 얘기한 이미지 파일을 통한 복구 방법을 적합하지 않다. 또한 대부분의 내장 시스템의 경우에는 사용하는 프로그램이 해당 시스템에 특성화되어 있으므로 해당 프로그램을 자유롭게 수정이 가능하다. 우리는 공유 메모리기법을 이용해서 이 두 가지 조건을 충족하도록 하였다. 공유 메모리는 한 프로세스에서만 국한이 되는 메모리 영역이 아니므로 여러 프로세스들이 동시에 접근이 가능하다. 이런 공유 메모리를 이용할 경우에는 프로세스 사라지더라도 메모리에 복구가 필요 되는 프로세스의 정보는 계속 유지가 되게 된다. 이 방법을 통해서 자신이 복구 시에 필요로 하는 정보만을 공유메모리에 저장하게 된다. 또한 저장 후에 복구 시에도 기존 프로세스를 재실행 후에는 앞에서 얘기한 일반적인 정보가 아닌 프로세스가 필요한 정보만을 복구하므로 인해서 빠른 복구가 가능하게 된다.

4. 구현

- 프로세스에 발생하는 크래쉬 감지

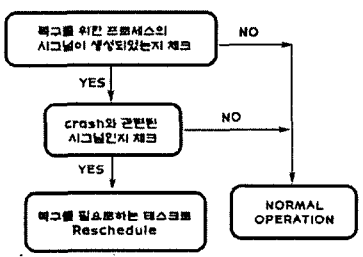
크래쉬 감지 부분은 크게 두 부분으로 나눌 수 있다. 실제 복구를 필요로 하는 프로세스에게 시그널을 보내는 부분과 복구를 필요로 하는 프로세스가 실제로 해당 시그널을 받아서 시그널 핸들러를 수행하도록 하는 부분이다.

첫 번째 부분인 복구를 필요로 하는 프로세스에게 시그널을 보내는 부분이다. 이 역할을 수행하는 함수는 아래와 표와 같이 네 개의 함수가 존재한다.

```
int send_sig_info(int, struct siginfo *, struct task_struct *);
int force_sig_info(int, struct siginfo *, struct task_struct *);
int send_sig(int, struct task_struct *, int);
int force sig(int, struct task_struct *, int);
```

표 3 시그널을 보내는 함수들(kernel/signal.h)

위에 네 함수 모두 체크하는 것이 아니라 실제로 나머지 세 함수는 send_sig_info를 호출하게 된다. 따라서 send_sig_info 함수만을 제어해 주면 된다. 여기서는 실제로 복구를 필요로 하는 프로세스에 크래쉬 관련 시그널들(표 1)이 보낼 시에는 해당 프로세스에게 확인 후에 참일 경우에 태스크 교환(task switching)이 일어나도록 구현이 되어 있다. 기존 논문에 얘기하는 구현으로는 여러 프로세스들이 동작 시에는 원하는 프로세스로 복귀시점이 늦어질 수 있다. 따라서 좀더 빠른



복구를 위해서는 크래쉬 시그널이 발생하자마자 바로 복구를 원하는 프로세스를 재스케쥴링하게 된다.

그림 1 signal send 부분에 흐름도

두 번째는 복구를 필요로 하는 프로세스가 실제로 해당 시그널을 받아서 시그널 핸들러를 수행하도록 하는 부분이다. do_signal 함수에서는 실행중인 프로세스(current process)에게 배달된 시그널에 대해서 처리를 하게 된다. 이 시점에서 복구를 원하는 프로세스일 경우이면서 크래쉬 관련 시그널일 시에는 해당 시그널 핸들러를 수행하는 대신 복구를 위한 내용이 있는 핸들러를 수행하도록 한다.

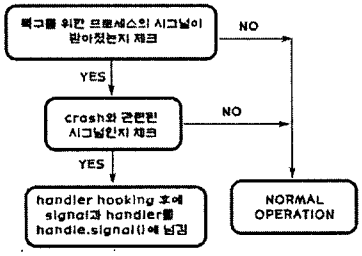


그림 2 signal delivery부분에 흐름도

- 프로세스를 복구

앞부분까지는 시스템 레벨에서 구현한 부분이다. 이 부분부터는 유저 레벨에서는 구현한 부분에 해당된다. 앞에서의 크래쉬 감지 부분에서 감지 후에 크래쉬와 관련된 시그널 핸들러를 호출하지 않고 특정 시스널에 맵핑된 복구를 위한 핸들러를 호출한다. 복구 부분에서는 미리 시그널 등록 함수를 통해서 특정 시그널에 복구 핸들러를 등록해 놓는다. 복구 핸들러함수에서는 execve함수를 통해서 프로세스를 재시작하게 한다. 그런 후에는 공유 메모리 함수를 호출하여 기존에 상태를 복구하게 된다.

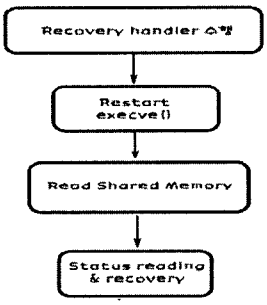


그림 3 복구 부분 흐름도

5. 실험

실험 시스템 사양은 CPU 800MHz, RAM 192MB, 리눅스 커널 버전 2.4.19이다. 시그널이 발생하고 이를 받아서 핸들러를 수행하고 다시 복구가 동작하는 데까지 걸리는 시간은 상당히 짧다. 기존 리눅스에 제공하는 jiffies값을 가지고 측정이 어렵다. 따라서 좀더 정밀한 시간간격을 측정하기 위해서 인텔(x86) 아키텍처에서 제공하는 TSC(Time Stamp Counter)를 이용하였다. 실행 파일은 계속적으로 숫자를 화면에 출력하는 프로그램을 사용하였다. 숫자 정보는 공유메모리에 저장을 하였다.

구간	평균 시간(usec)
감지구간(signal delivery phase)	10
복구구간(execve execution phase)	304
총 구간(total phase)	314

표 4 sigsegv실행 후에 time측정결과

위에 결과에서 볼 수 있듯이 시그널이 감지가 되어서 복구를 실행하기 전까지는 평균적으로 10usec이내에 되는 것을 볼 수 있다. 따라서 데몬 형식으로 감지할 경우에 task switch문제로 인해서 최소한 msec단위의 시간이 걸리는데 비해서는 훨씬 나은 성능을 보이고 있다. 이 기존 논문에서는 시그널이 감지가 되어서 복구를 실행하기 전까지의 시간이 약 6usec정도를 보이고 있다. 시스템 사양은 CPU III 1.2GHz, RAM 512MB이다. 결과는 기존 논문에 시스템 사양은 CPU클럭이 1.2GHz이므로 우리 시스템에 적용할 시에 해당 시간이 약 9usec전후 가지게 된다. 거의 비슷한 결과를 가지게 된다. 반면에 여러 프로세스가 작동할 시에는 대한 내용의 설명이 존재하지 않는다. 현 논문에서 제안 해당 프로세스 크래쉬를 일으킬 경우에는 감지 시에 재스케쥴링하므로 바로 복구 시그널 핸들러가 수행이 된다. 따라서 여러 프로세스가 동작 시에도 거의 비슷한 수준의 감지 시간이 걸리게 된다. execve의 실행 시간은 어떤 프로세스를 복구하느냐에 따라서 실행시간 자체가 달라지므로 크게 의미가 있지는 않다.

테스트 프로그램 파일의 사이즈가 4822bytes인데 uclik[7]로 생성하는 하나의 복구를 위한 이미지 파일 사이즈는 71391bytes를 가진다. 실행 파일 사이즈보다 큰 결과를 가져온다. 반면에 공유메모리를 사용할 경우에는 숫자 정보를 저장할 integer사이즈인 4bytes만을 가지면 복구가 가능하다. 따라서 내장 시스템에 경우에는 공유메모리 방식을 사용하는 것이 훨씬 더 합리적인 것을 알 수 있다.

6. 결론

이번 연구는 실시간 내장 시스템에서 좀더 효율적인 복구를 통하여 시스템의 실시간성을 향상시키는 것을 목표로 구현되었다. 제안된 공유 메모리를 이용한 프로세스 복구 기법[1]은 기존에 제안된 체크 포인팅(check pointing)과 롤백(rollback)을 이용한 기법보다 적은 메모리 공간을 차지함으로써 내장 시스템의 주요한 성격인 자원의 제약적인 상황에 적합하게 구현되었다. 또한 크래쉬 등지 시그널이 발생하였을 때, 에러를 감지한 후에 시그널 후킹(hooking) 기법[1]과 프로세스 스케줄 순서를 조정하는 방법을 통해 좀더 빠른 시그널 핸들러를 구현함으로써 기존의 에러 감지 기법보다 더욱 빠르게 에러를 감지하게 되었다. 이것으로 인해 시스템에 크래쉬가 발생하였을 때 향상된 실시간성을 통하여 시스템이 데드라인에 접근하는 확률을 크게 떨어뜨렸다.

앞으로 개선할 부분으로는 복구 프로세스를 크래쉬 관련 시그널이 생성시에는 스케줄 순서를 조정함으로써 기존 프로세스들에 미치는 영향에 대해서 좀더 보완이 필요할 것이다.

7. 참조 문헌

[1] Kihwal Lee and Lui Sha, "Process Resurrection: A Fast Recovery Mechanism for Real-Time Embedded Systems." *Proceedings of the 11th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'05)*

[2] Kuo-Feng Ssu; Fuchs, W.K.; Jiau, H.C. "Process recovery in heterogeneous systems." *Computers, IEEE Transactions on Volume 52, Issue 2,* Feb. 2003 Page(s):126 – 138

[3] G. Candea and A. Fox, "Recursive Restartability: Turning the Reboot Sledgehammer into a Scalpel," *8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, chloss Elmau, Germany,May 2001.

[4] R. Kirner, P. Puschner, and I. Wenzel, "Measurement-Based Worst-Case Execution Time Analysis," *Proceedings of 4th Euromicro International Workshop on WCET nalysis*, June, 2004.

[5] Eric Roman "A Suvey of Checkpoint/Restart Implementations"

[6] The Home of Checkpointing Packages http://checkpointing.org

[7] UCLiK: Unconstrained Checkpointing in the Linux Kernel http://www.cise.ufl.edu/~mfoster/research/uclik/uclik.htm

[8] Libckpt : A Portable Checkpointer for Unix http://www.cs.utk.edu/~plank/plank/www/libckpt.html